Caractérisation analytique et optimisation de codes source-canal conjoints

Amadou Tidiane Diallo

UNIVERSITÉ PARIS SUD

Comprendre le monde, construire l'avenir®

L2S

UNIVERSITE PARIS-SUD

DISCIPLINE : PHYSIQUE

ÉCOLE DOCTORALE

Sciences et Technologies de l'Information des Télécommunications et des Systèmes

Laboratoire des Signaux et Système

THÈSE DE DOCTORAT

Soutenue le 1er octobre 2012

Amadou Tidiane DIALLO

Caractérisation Analytique et Optimisation de Codes Source-Canal Conjoints

Composition du jury

M. Yannis MANOUSSAKIS — LRI, Université Paris-Sud — Président du jury

M. Charly POULLIAT — INP-ENSEEIHT, Université de Toulouse — Rapporteur

Mme Serap SAVARI — Texas A&M University — Rapporteur

Mme Catherine LAMY-BERGOT — Thales Communications — Examinateur

M. Michel KIEFFER — L2S, Université Paris-Sud — Directeur de thèse

M. Claudio WEIDMANN — ETIS-ENSEA, Université de Cergy-Pontoise — Co-directeur de thèse

Remerciements

Voilà pour moi, c'est la fin de la thèse. Rapport bouclé, soutenue effectuée ; quel soulagement ⌣̈.

Cependant, en jetant un coup d'oeil sur toutes ces années de thèse, il me parait de manière indéniable que ce travail a été en partie possible grâce à la présence d'un bon nombre de personnes d'une générosité exceptionnelle. Et ce paragraphe est dédié à ces personnes.

Mes premiers remerciements vont tout naturellement à mon directeur de thèse Michel Kieffer qui a accepté de me choisir en tant que doctorant. Sa disponibilité, ses compétences et son encouragement m'ont été d'un grand service pour mener à bien ces travaux. J'ai pu apprécier sa rigueur, sa passion des sciences et sa dimension humaine qui me seront d'une grande inspiration. J'en profite ici pour lui exprimer toute ma profonde gratitude.

Je remercie tout chaleureusement mon co-directeur de thèse Claudio Weidmann qui n'a ménagé aucun effort durant toutes ces années pour le bon déroulement de cette thèse. Les conseils prodigués et les orientations qu'il a suggérées ont donné une autre dimension à cette thèse. Un grand merci pour m'avoir fait découvrir la superbe ville de Vienne et un grand merci pour sa disponibilité. J'ai été très honoré par son encadrement.

Mes remerciements vont également à Pierre Duhamel la personne par la quelle mon aventure de doctorat a commencé au L2S. Merci de m'avoir fait confiance pour le master2R Res-Tel, de m'avoir permis de trouver une thèse et de s'assurer que je termine cette thèse dans des bonnes conditions. Les discussions menées ensemble ont su donner des nouvelles orientations à cette thèse. Je lui exprime ici toute ma reconnaissance.

Je remercie très sincèrement Charly Poulliat et Serap Savari d'avoir accepté de rapporter ma thèse. Les remarques apportées ont permis d'améliorer la lisibilité du manuscrit.

Je suis très honoré que Catherine Lamy-Bergot et Yannis Manoussakis aient accepté d'examiner mes travaux de thèse. Merci à Yannis Manoussakis pour avoir présidé mon jury.

Je tiens également à remercier toute l'équipe du L2S que j'ai côtoyée durant

toutes ces années. Un grand merci à José et Veronica pour leur amitié sincère. Merci à Lana, Sofiane, Khaled, Cedric, Thang qui ont partagé le même bureau et qui ont su me tenir compagnie sans le moindre encombre. Une grande pensée pour Alex, Francesca, Elsa, Mael, Nabil, Hamidou Tembiné, Cagatay, Hacheme, Diarra, Odette pour leur disponibilité et leur gentillesse exceptionnelles. Mes amitiés vont également à Zeina, Neila, Samir, Ziad, Brice, Laurie, Aurelia, Mathieu, Pierre Gerold, François Meriaux, Benjamin, Vineeth, Olivier, Anna, Jean François, Ngoc, Dinh Tuan, Xuan Thang, Duy, Patrice, Samson, Hassan Hijazi, Chengfang avec qui j'ai passé des moments inoubliables.

Je veux remercier aussi quelques personnes qui me sont proches Joseph Abissi, Cheick Kébé, Babacar Diop, Gondia Seck, Issa Ngom, Abou Ndiongue, Harouna Ly, Samson et Vitia Dirave pour les discussions et les bons moments passés ensemble.

Un grand merci à toutes ma belle famille : la famille Ponnoussamy (beau-père, belle-mère Regina et beau-frère Prospère) et la famille Leçon (Luc, Selvy, Wilbert, Willem, Joseph, Pouvena) pour leur soutient.

Toute ma reconnaissance à ma première famille d'accueil en France que sont Yaya, Tacko et toute la famille Ndiaye.

Enfin un grand merci pour ma femme Pounida et ma fille Éléonore Abiramy qui m'ont soutenu durant toutes ces années et avec qui la vie prend tout son sens. Je ne saurai exprimer toute ma gratitude à feu mon père Mamadou, à ma mère Djeynaba Sy, à tous mes frères (Abou, Saïdou) et toutes mes sœurs (Aïssata, Mariam, Alimata, Aminata, Maïmouna, Khardiatou) pour la vie. Je leur dédie cette thèse.

Résumé

Les *codes source-canal conjoints* sont des codes réalisant simultanément une compression de données et une protection du train binaire généré par rapport à d'éventuelles erreurs de transmission. Ces codes sont non-linéaires, comme la plupart des codes de source. Leur intérêt potentiel est d'offrir de bonnes performances en termes de compression et de correction d'erreur pour des longueurs de codes réduites.

La performance d'un *code de source* se mesure par la différence entre l'*entropie* de la source à compresser et le *nombre moyen de bits* nécessaire pour coder un symbole de cette source. La performance d'un *code de canal* se mesure par la *distance minimale* entre mots de codes ou entre suite de mots de codes, et plus généralement à l'aide du *spectre des distances*. Les codes classiques disposent d'outils pour évaluer efficacement ces critères de performance. Par ailleurs, la synthèse de bons codes de source ou de bons codes de canal est un domaine largement exploré depuis les travaux de Shannon. Par contre des outils analogues pour des codes source-canal conjoints, tant pour l'évaluation de performance que pour la synthèse de bons codes restaient à développer, même si certaines propositions ont déjà été faites dans le passé.

Cette thèse s'intéresse à la famille des codes source-canal conjoints pouvant être décrits par des *automates* possédant un nombre fini d'états. Les *codes quasi-arithmétiques correcteurs d'erreurs* et les *codes à longueurs variables correcteurs d'erreurs* font partie de cette famille. La manière dont un automate peut être obtenu pour un code donné est rappelée.

A partir d'un automate, il est possible de construire un *graphe produit* permettant de décrire toutes les paires de chemins divergeant d'un état et convergeant vers le même ou un autre état. Nous avons montré que grâce à l'algorithme de Dijkstra, il est alors possible d'évaluer la distance libre d'un code conjoint avec une complexité polynomiale.

Pour les codes à longueurs variables correcteurs d'erreurs, nous avons proposé des bornes supplémentaires, faciles à évaluer. Ces bornes constituent des extensions des bornes de Plotkin et de Heller aux codes à longueurs variables. Des bornes peuvent également être déduites du graphe produit associé à un code dont seule une partie des mots de code a été spécifiée.

Ces outils pour borner ou évaluer exactement la distance libre d'un code conjoint permettent de réaliser la synthèse de codes ayant des bonnes propriétés de distance pour une redondance donnée ou minimisant la redondance pour une distance libre donnée. Notre approche consiste à organiser la recherche de bons codes source-canal conjoints à l'aide d'arbres. La racine de l'arbre correspond à un code dont aucun bit n'est spécifié, les feuilles à des codes dont tous les bits sont spécifiés, et les noeuds intermédiaires à des codes partiellement spécifiés. Lors d'un déplacement de la racine vers les feuilles de l'arbre, les bornes supérieures sur la distance libre décroissent, tandis que les bornes inférieures croissent. Ceci permet d'appliquer un algorithme de type branch-and-prune pour trouver le code avec la plus grande distance libre, sans avoir à explorer tout l'arbre contenant les codes.

L'approche proposée a permis la construction de codes conjoints pour les 26 lettres de l'alphabet. Comparé à un schéma tandem équivalent (code de source suivi d'un code convolutif), les codes obtenus ont des performances comparables (taux de codage, distance libre) tout en étant moins complexes en terme de nombre d'états du décodeur.

Plusieurs extensions de ces travaux sont en cours : 1) synthèse de codes à longueurs variables correcteurs d'erreurs formalisé comme un problème de programmation linéaire mixte sur les entiers ; 2) exploration à l'aide d'un algorithme de type A* de l'espace des codes à longueurs variables correcteur d'erreurs.

Abstract

Joint source-channel codes are codes simultaneously providing data compression and protection of the generated bitstream from transmission errors. These codes are non-linear, as most source codes. Their potential is to offer good performance in terms of compression and error-correction for reduced code lengths.

The performance of a *source code* is measured by the difference between the *entropy* of the source to be compressed and the *average number of bits* needed to encode a symbol of this source. The performance of a *channel code* is measured by the *minimum distance* between codewords or sequences of codewords, and more generally with the *distance spectrum*. The classic codes have tools to effectively evaluate these performance criteria. Furthermore, the design of good source codes or good channel codes is a largely explored since the work of Shannon. But, similar tools for joint source-channel codes, for performances evaluation or for design good codes remained to develop, although some proposals have been made in the past.

This thesis focuses on the family of joint source-channel codes that can be described by *automata* with a finite number of states. *Error-correcting quasi-arithmetic codes* and *error-correcting variable-length codes* are part of this family. The way to construct an automaton for a given code is recalled.

From an automaton, it is possible to construct a *product graph* for describing all pairs of paths diverging from some state and converging to the same or another state. We have shown that, using Dijkstra's algorithm, it is possible to evaluate the free distance of a joint code with polynomial complexity.

For errors-correcting variable-length codes, we proposed additional *bounds* that are easy to evaluate. These bounds are extensions of Plotkin and Heller bounds to variable-length codes. Bounds can also be deduced from the product graph associated to a code, in which only a part of code words is specified.

These tools to accurately assess or bound the free distance of a joint code allow the design of codes with good distance properties for a given redundancy or minimizing redundancy for a given free distance. Our approach is to organize the search for good joint source-channel codes with trees. The root of the tree corresponds to a code in which no bit is specified, the leaves of codes in which all bits are specified, and the intermediate nodes to partially specified codes. When moving from the root

to the leaves of the tree, the upper bound on the free distance decreases, while the lower bound grows. This allows application of an algorithm such as *branch-and-prune* for finding the code with the largest free distance, without having to explore the whole tree containing the codes.

The proposed approach has allowed the construction of joint codes for the 26 letters of the alphabet. Compared to an equivalent tandem scheme (source code followed by a convolutional code), the codes obtained have comparable performance (rate coding, free distance) while being less complex in terms of the number of states of the decoder.

Several extensions of this work are in progress : 1) synthesis of error-correcting variable-length codes formalized as a mixed linear programming problem on intergers, 2) Explore the search space of error-correcting variable-length codes using an algorithm such as A* algorithm .

Glossaire

AC	Codage Arithmétique *Arithmetic Coding (AC)*
C(n,k,d)	Code en bloc linéaire de longueur de bloc n, 2^k mots de code et distance minimale de d
CAE	Codage Arithmétique Entier *Integer Arithmetic Coding (IAC)*
CBS	Canal Binaire Symétrique *Binary Symmetric Channel (BSC)*
CC	Code Convolutif *Convolutional Code (CC)*
CLV	Code à Longueur Variable *Variable-Length Code (VLC)*
CLVR	Code à Longueur Variable Réversible *Reversible Variable-Length Code (RVLC)*
CEF	Codeur à États Finis *Finite-State Encoder (FSE)*
$\mathbf{d}_{\text{min}}$	La distance minimale
$\mathbf{d}_{\text{libre}}$	La distance libre *Free Distance* ($\mathbf{d}_{\text{free}}$)
GDP	Graphe des Distances entre Paires *Pairwise Distance Graph (PDG)*
GPM	Graphe Produit Modifié *Modified Product Graph (MPG)*
LV	Longueur Variable *Variable-Length (VL)*
MCT	Modulation Codée sur Treillis *Trellis-Coded Modulation (TCM)*
MEF	Machine à états finis *Finite-State Machine (FSM)*
MV	Maximum de Vraisemblance *Maximum Likelihood (ML)*
CQA	Code quasi arithmétique *Quasi arithmetic coding (QAC)*
SCC	Source-Canal Conjoint *Joint Source-Channel (JSC)*
SI	Symbole Interdit *Forbidden Symbol* (FS)
SIM	Symbole Interdit Multiple *Multiple Forbidden Symbol (MFS)*

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Introduction

L'objectif principal de tout système de communication numérique est de transmettre une information d'une source qui génère cette information à un destinataire avec la plus grande fidélité possible. La figure 1.1 illustre le diagramme fonctionnel et quelques éléments constitutifs d'un système de communication numérique. La sortie de la source peut être numérique (par exemple la sortie d'un ordinateur) ou analogique (par exemple un signal audio, vidéo ...) dans ce dernier cas une compression avec perte est introduite pour numériser l'information.

Dans un système de communication numérique, le message produit par la source est converti en séquences de bits. Idéalement, on souhaiterait représenter la sortie de la source avec le plus petit nombre de bits possible. Dans [Sha48], Shannon montre que le nombre minimal de bits à utiliser pour représenter un symbole de la source sans perte d'information est l'*entropie* de la source. Donc, on cherche une représentation efficace de la sortie de la source qui aboutit à très peu (ou pas) de redondance (la différence entre le nombre bit utilisé pour représenter un symbole généré par la source et l'entropie).

Le processus de conversion efficace de la sortie de la source en séquence binaire se nomme *codage de source* ou *compression de données*. L'efficacité de la compression d'un codeur de source est mesurée par le rapport entre *la longueur moyenne* du code et l'entropie de la source [CT91].

Comme exemple de codage de source, on peut citer les codes à longueur variable

[Huf52] et les codes arithmétiques (CA) [Pas76, Ris76] qui sont utilisés dans de nombreux schémas pour la compression de texte, de musique, d'image, ou de vidéo [Say05]. Cette utilisation est due à leur bonne efficacité de compression et leur implémentation facile.

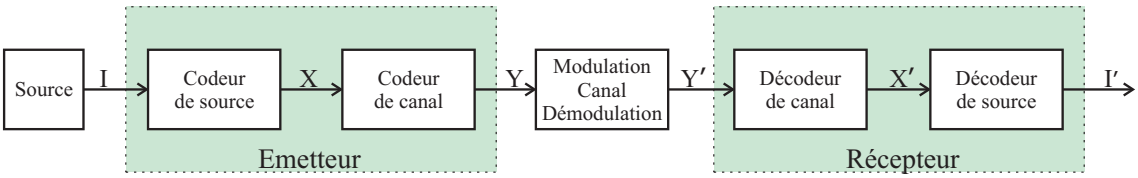


FIGURE 1.1 – Schéma classique simplifié d'une communication numérique

Néanmoins, le fait que les séquences de bits générées par un codeur de source soient à longueur variable les rend particulièrement vulnérables à l'égard d'erreurs de transmission introduites par le canal de transmission. Ainsi, les données encodées en utilisant un codeur de source sont habituellement protégées en les faisant passer à travers un *codeur de canal*.

Le but de codeur de canal est d'introduire de manière contrôlée de la redondance dans la séquence binaire qui sera utilisée par le *décodeur de canal* introduit au niveau du récepteur pour venir à bout des effets du bruit et des interférences lors de la transmission du signal à travers le canal. Ainsi, la redondance dans la séquence aide le décodeur de canal au niveau du récepteur pour estimer les bits générés par le codeur de source.

La capacité de correction d'erreurs d'un code de canal peut être prédite par la *borne de l'union des événements d'erreurs* en utilisant ses *propriétés de distance* c'est-à-dire sa *distance libre* et son *spectre de distances*, voir [VO79, chapitre 4]. On peut citer les *codes en blocs* et les *codes convolutifs* comme exemples de codes de canal.

Les estimées des bits fournies par le décodeur de canal sont ensuite utilisées par le *décodeur de source* pour fournir une estimée des symboles d'information générés par la source.

Le processus de modulation et de démodulation est transparent pour nous.

Un tel *schéma de codage séparé (tandem)* est motivé par le principe de séparation de Shannon [Sha48, CT91] qui formule que le codage de source et de canal peuvent être optimisés séparément sans perte d'optimalité par rapport à un schéma

conjoint réalisant les deux opérations simultanément mais avec une complexité potentiellement plus importante (pour le schéma tandem). Cependant, ce résultat a été obtenu en faisant des hypothèses d'un canal stationnaire avec des caractéristiques bien connues, ce qui est rarement le cas avec les systèmes de communication sans fil et des mots de code de longueur infinie, ce qui n'est pas réaliste pour des applications avec des contraintes de délais d'acheminement. Les schémas tandems avec des longueurs de mot de code finies et/ou avec un canal non stationnaire sont donc sous-optimaux.

Pour une complexité et un délai fixés, les *codes source-canal conjoints* pourraient être meilleurs que les schémas tandem, voir [ZAC06]. L'objectif de la synthèse de codes conjoints est de trouver des codes plus performants (soit en terme de correction d'erreurs pour une complexité donnée soit en terme de complexité pour un taux d'erreurs fixé) que les codes tandem lorsque la longueur du code est contrainte [ZAC06] (voir figure 1.2). Le problème réside dans la construction des codes conjoints optimaux, et le but de cette thèse est de proposer des méthodes de construction efficaces de codes conjoints performants.

Les codes conjoints peuvent être construits à partir des codes à longueur variable conduisant aux *codes à longueur variable conjoints* et à partir des codes arithmétiques (CAs) conduisant aux *codes arithmétiques conjoints*.

Comme pour le schéma tandem, l'efficacité de la compression des codes conjoints est mesurée par le rapport entre *la longueur moyenne* du code et l'entropie de la source [CT91] alors que sa capacité de correction d'erreurs peut être prédite par la *borne de l'union* en utilisant ses *propriétés de distance* c'est-à-dire sa *distance libre* et son *spectre de distances*, voir [VO79].

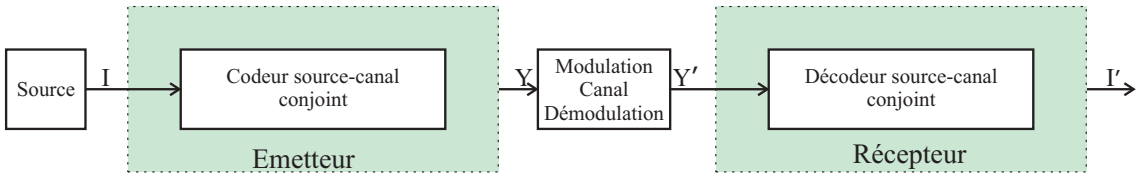


FIGURE 1.2 – Schéma conjoints simplifié d'une communication numérique

Cette thèse se focalise sur l'optimisation des codes à longueur variable conjoints et des codes arithmétiques conjoints en maximisant la distance libre, c'est-à-dire la plus petite distance entre suite de mots de code de même longueur cumulée. Notre

approche consiste à explorer deux sous-ensembles de l'ensemble des *codeurs à états finis* (CEFs) [Pir82] qui génèrent les *codes à états finis* que sont les codes à longueur variable conjoints et codes arithmétiques conjoints. Une condition préalable à une telle optimisation est la disponibilité d'outils efficaces pour évaluer les propriétés de distance des codes à états finis, ce qui constitue la première contribution de cette thèse (voir le chapitre 3). Cette optimisation nécessite aussi des outils pour borner la distance libre des codes conjoints, ce qui constitue la deuxième contribution de cette thèse (voir le chapitre 4).

Le paragraphe 1.2 présente quelques outils disponibles dans la littérature pour l'évaluation des propriétés de distance et leur limitations. Le paragraphe 1.3 fait un bref rappel sur quelques méthodes d'optimisation des codes conjoints disponibles dans la littérature. Le paragraphe 1.4 présente les contributions principales de cette thèse et la structure du rapport.

1.2 Évaluation des propriétés de distances

La distance de Hamming, $d_{\mathrm{H}}(\boldsymbol{x}, \boldsymbol{y})$ entre deux séquences binaires $\boldsymbol{x}$ et $\boldsymbol{y}$ de même longueur est le nombre de bits différents entre entre $\boldsymbol{x}$ et $\boldsymbol{y}$. C'est le nombre de bits qu'il faudrait changer pour confondre $\boldsymbol{x}$ et $\boldsymbol{y}$. Pour un code composé d'un ensemble de séquences de longueur (semi-)infinie (les mots de code), la *distance libre* d_{libre} est la plus petite distance de Hamming entre toutes les paires de séquences distinctes possibles. La distance libre détermine la capacité de détection et de correction d'erreurs d'un code. En gros, un code de *distance libre* d^0_{libre} pourra détecter $d^0_{\text{libre}} - 1$ erreurs dans une séquence et corriger $\left\lfloor \frac{d^0_{\text{libre}}-1}{2} \right\rfloor$ erreurs, où la fonction $\lfloor x \rfloor$ arrondit vers le plus grand entier inférieur ou égal à x.

Les premiers outils pour l'évaluation des propriétés de distance considèrent les codes à états finis linéaires tel que les codes convolutifs (CCs) [Eli55]. Dans [Vit71], Viterbi calcule la *fonction de transfert* sur le *digramme d'état* des CCs pour obtenir le spectre de distance (c'est-à-dire à quelle distance se trouvent les mots de code les uns des autres) et en déduire la distance libre. Dans [Ast86] une variante de l'algorithme de plus court chemin de Dijkstra est appliquée sur le digramme d'états des CCs pour calculer la distance libre sans générer le spectre de distance. Plus tard, [CJ89] propose un algorithme rapide en arbre pour calculer la spectre de distances

des CCs. Toutes ces méthodes ont des complexités linéaires en nombre d'états grâce à la linéarité des CCs.

Pour les codes à états finis non linéaires, toutes les paires de mots de code doivent être comparées pour calculer la distance libre. Pour les *codes à distance euclidienne* générés par des *modulations codées sur treillis* [Ung82], [Big84] utilise le *graphe produit* associé au codeur à états finis. Ceci permet de calculer le spectre de distance des modulations codées sur treillis dans le domaine code et d'en déduire la distance libre. Pour un code à états finis avec 2^ν états, un *treillis produit* avec $2^{2\nu}$ états est nécessaire pour ces évaluations [Big84]. Pour la classe des codes à états finis *géométriquement uniforme* [For91], qui incluent certaines modulations codées sur treillis, une *fonction génératrice modifiée* sur un diagramme d'états avec seulement 2^ν états est suffisant pour calculer le spectre [ZW87].

Toutes les techniques décrites précédemment sont pour les codes à *longueur fixe* (*taux de codage fixe*), plus précisément pour les codeurs à états finis définis par des graphes où toutes les transitions ont des étiquettes d'entrée de même longueur k ainsi que des étiquettes de sorties de même longueur n, comme dans le cas, par exemple, des CCs de taux de codage k/n.

Les propriétés de distance des codes à longueur variable conjoints ont été évaluées pour la première fois dans [BF94, But95], où une borne inférieure de leur distance libre et un algorithme exhaustif (de complexité exponentielle) pour leur spectre de distance ont été proposés. Des graphes similaires aux graphes utilisés pour les propriétés de distance des codes treillis à longueur fixe ont aussi été utilisés dans le contexte des codes à longueur variable, mais pour évaluer d'autres propriétés. Par exemple, [Eve64] définit un *graphe de test* qui est le graphe produit qui ne représente que les paires de chemins qui sont à une distance de Hamming nulle pour définir un *test de synchronisabilité* des codes à longueur variable. L'*error-state diagram* introduit par [MR85] est un graphe produit qui représente les paires de chemins à une distance de Hamming de 1 pour étudier les propriétés de resynchronisation du décodeur, suite à l'occurrence d'une erreur bit sur la séquence codée. Dans [MJG08], ces résultats ont été étendus au cas des codes arithmétiques conjoints.

L'évaluation des propriétés de distances des codes à états finis non linéaires correspondants aux codes arithmétiques conjoints est plus complexe que pour les codes à longueur variable conjoints. Ceci est du au fait que les codes à longueur variable

conjoints n'ont qu'un seul état dans lequel les chemins divergent et convergent, alors que les codes arithmétiques conjoints peuvent avoir plusieurs états de ce type. Les premiers outils analytiques pour les codes arithmétiques conjoints ont été proposés dans [BJWK08], où la distance libre est évaluée avec une complexité polynomiale. Une approximation du spectre de distances est également obtenue mais avec une complexité exponentielle comme dans [But95]. Dans [WK10], les propriétés de distance des *codes à états finis à longueur variable* générés par des *codeurs à états finis à longueur variable* (CEF-LVs) sont considérées. Une méthode matricielle de complexité polynomiale est proposée pour le calcul exact du spectre de distances ou pour le calcul d'une borne supérieure sur le spectre. Les définitions d'un codeur à états finis à longueur variable et de codes à états finis à longueur variable seront rappelées au chapitre 2.

Tous les outils cités précédemment pour le calcul des propriétés de distance des codes à longueur variable conjoints et des codes arithmétiques conjoints ont une bonne efficacité, mais restent très complexes pour la recherche de codes (voir le chapitre 3).

Pour cette raison, dans le chapitre 3, nous allons d'abord généraliser les méthodes proposées par [Big84] et [Ast86] à tous les codes à états finis afin de pourvoir évaluer efficacement les propriétés de distance avec une complexité réduite par rapport à celle des techniques déjà présentent dans la littérature (pour les codes à état finis à longueur variables). Un graphe produit inspiré de celui présenté dans [Big84] est proposé pour les codeurs à états finis en général et est simplifié pour obtenir deux graphes : le *graphe produit modifié* (GPM) qui permet de calculer le spectre de distance dans le domaine code en utilisant une approche de la fonction de transfert en appliquant la formule du gain de Mason [Mas56] ; et le *graphe des distances entre paires* qui permet de calculer la distance libre du code à états finis en appliquant l'algorithme de Dijkstra comme dans [Ast86] sans calculer le spectre de distance en entier [DWK11]. Cette approche est moins complexe que l'approche proposée dans [BJWK08] (voir le paragraphe 3.2 du chapitre 3).

Ensuite, notre objectif est d'appliquer ces outils d'évaluation de distance pour une optimisation efficace des codes à longueur variable conjoints et codes arithmétiques conjoints.

1.3 Synthèse de codes conjoints optimisés

1.3.1 Codes à longueur variable conjoints

Les méthodes de constructions des codes à longueur variable conjoints peuvent être classées selon la manière dont les conditions de préfixe ou de suffixe, les propriétés de distances, la longueur moyenne des codes, *etc.* entrent dans le processus de construction.

Une première famille de méthodes réalise une construction progressive des codes en garantissant certaines propriétés à chaque étape de la construction. Une seconde famille considère une liste exhaustive de codes dont les propriétés sont examinées pour trouver le meilleur.

Les codes *bidirectionnels* ou *codes à longueur variable réversibles* (CLVRs) introduits par [FK90] forment une classe de codes à longueur variable conjoints qui ont été très étudiés. Ces codes sont décodables dans les deux directions avant et arrière, c'est-à-dire qu'ils sont préfixes et suffixes (*fix-free*). La construction des CLVRs tente en général de minimiser la longueur moyenne du code, voir par exemple, [TWM95, TW01, LV03, TC03, WYH04, LWC08, HWH10] sans tenir compte des contraintes sur la distance libre.

Plusieurs techniques de construction de codes bidirectionnels ont été étendues à la construction des codes à longueur variable avec des plus grandes distances libres. Une simple extension de [TW01] aux codes à longueur variable avec des distances libres supérieures ou égale à deux est fournies dans [LV02, LWC08]. Ceci est fait en imposant une *distance de bloc* (c'est-à-dire la distance minimale entre les mots de code de même longueur voir, [BF94, But95]) minimale pour les codes à longueur variable. Dans [HWH10], la recherche optimale de CLVRs en terme de minimisation de la longueur moyenne des mots de code est traitée comme une recherche en arbre, sur lequel un algorithme A* est appliqué. L'inconvénient majeur de ces méthodes est que la condition de *fix-free* réduit considérablement l'espace de recherche lorsque les longueurs des mots de code sont fixées, et donc limite la possibilité de trouver de bons codes en terme par exemple de distance libre.

Deux techniques de constructions de codes à longueur variable avec une distance libre donnée ont été proposées dans [BF94, But95]. La première commence avec un code canal, dont les mots de code sont raccourcis tout en préservant une certaine

propriété de distance. La seconde construit progressivement les mots de code tout en assurant une bonne *distance par bloc* et une *distance divergente*, c'est-à-dire la distance entre les préfixes, qui est une borne inférieure de la distance minimale entre les mots de code. Les mots de code obtenus ont des longueurs qui ne sont pas nécessairement adaptées aux statistiques de la source. De plus, si le mot de code le plus court a une longueur inférieure à la distance libre désirée alors aucun code ne peut être trouvé avec ces techniques.

Ces techniques ont été améliorées en terme de complexité par [LP03] et [WYH04]. Plus tard, [TK09] construit des codes pour un décodage itératif en imposant des mots de code de poids impairs assurant une distance libre minimale de deux. Néanmoins, cette technique n'est pas applicable sur des codes avec des distances plus grandes. Dans [MH09] un algorithme générique basé sur la construction de codes qui maximise les performances de compression tout en satisfaisant une borne inférieure de la distance libre est proposé. Cette approche complète la borne inférieure de la distance libre de [But95, BF00] avec un terme de correction incluant une mesure de dissimilarité des mots de codes, limitant la borne de la distance libre et les probabilités d'occurrence des mots de code. Une approche différente de celle proposée dans [HWH10] et qui consiste à formuler le problème de synthèse de code comme un problème de satisfaction de contraintes sur des binaires est proposée dans [Sav09, AKS10]. Cette approche permet de tenir compte des contraintes sur les distances divergentes, convergentes et par blocs entre les mots de code. Ceci permet d'obtenir les codes avec les propriétés de distances souhaitées (la borne inférieure est garantie d'être satisfaite), mais pas forcement le code avec une efficacité de codage optimale.

1.3.2 Codes arithmétiques conjoints

Pour les codes arithmétiques conjoints, la forme la plus commune introduit de la redondance dans le train binaire compressé au moyen d'un *symbole interdit*. Ce dernier n'est jamais émis par la source mais une probabilité non nulle lui est assignée lors de la subdivision de l'*intervalle de codage* durant le processus de codage [BCI+97]. Plus la probabilité du symbole interdit est grande, plus la redondance introduite et la robustesse contre les erreurs de transmission sont importantes. L'idée

est étendue dans [Say99], qui propose l'introduction de symboles interdits *multiples*.

Toutes ces techniques peuvent être appliquées aux *codage arithmétique en précision finie* [HV92] (c'est-à-dire un codage arithmétique où l'intervalle initial est un intervalle entier appartenant à un intervalle d'entiers défini par la précision du dispositif de codage (voir le chapitre 2) conduisant au code arithmétique conjoint en précision finie. Ainsi, le codage arithmétique conjoint en précision finie peut être défini par ces *paramètres caractéristiques* (les probabilités de la source, la précision avec laquelle le codage est réalisé et le taux de codage).

L'optimisation des codes arithmétiques conjoints en précision finie a été considérée dans [BJWK08] en supposant que la probabilité totale allouée au symbole interdit multiple et la probabilité de chaque symbole interdit individuelle sont indépendantes de l'état du codeur à états finis représentant le code arithmétique conjoint en précision finie (*indépendance vis-à-vis de l'état*). Cependant, la classe des codes arithmétiques conjoints en précision finie, lorsque les probabilités des symboles interdits sont allouées avec une indépendance vis-à-vis de l'état est significativement plus petite que la classe obtenue avec une allocation des probabilités des symboles interdits qui dépendent de l'état du codeur à états finis (*dépendance vis-à-vis de l'état*). Par conséquent, pour une redondance donnée, l'allocation des probabilités des symboles interdits avec une dépendance vis-à-vis de l'état pourrait conduire à des codes arithmétiques conjoints en précision finie plus robustes que ceux obtenus avec une allocation des probabilités des symboles interdits avec une indépendance vis-à-vis de l'état.

1.3.3 Proposition pour la construction de codes conjoints

La troisième contribution de cette thèse, décrite au chapitre 5, est de proposer des algorithmes pour une optimisation globale de la distance libre des codes source-canal conjoints [DWK09, DWK10, DWK11, DWK12]. Comme nous l'avons indiqué précédemment, cette optimisation se focalise sur deux sous-ensembles de codes source-canal conjoints : les codes à longueur variable conjoints et les codes arithmétiques conjoints.

Pour un ensemble de longueurs de mots de code fixé (pour les codes à longueur variable) ou pour des paramètres caractéristiques fixés (pour les codes arithmétiques),

cette thèse propose d'organiser l'ensemble des codes conjoints *préfixes* possibles de tel sorte qu'ils puissent être explorés avec un algorithme de type *branch-and-prune* pour trouver le(s) code(s) avec la meilleure distance libre.

Notre critère de recherche est la vraie valeur de la distance libre maximale possible (et non une borne de la distance libre). L'exécution de l'algorithme branch-and-prune exploite des bornes sur la distance libre de codes à états finis *incomplets* et *complets* définis dans le chapitre 4.

1.4 Contributions et structure du rapport

Trois principales contributions seront présentées dans cette thèse.

1. Une généralisation des méthodes établies sur graphe pour les codes à longueur fixes pour calculer la distance libre des codes à états finis à longueur variables générés par des codeurs à états finis à longueur variable.
2. Des nouvelles bornes sur la distances des codes à états finis à longueur variable qui sont utiles pour l'optimisation des codes.
3. Des méthodes de structuration du problème pour une optimisation de la distance libre des codes à longueur variable conjoints et des codes arithmétiques conjoints.

Le reste du rapport est structuré de la manière suivante.

Le chapitre 2 présente quelques notions sur les codes source-canal conjoints, plus précisément, il introduit les propriétés de distance de ces codes. Les propriétés de distances des codes de canal, tels que les codes en bloc et les codes convolutifs sont rappelées au début du chapitre.

Le chapitre 3 introduit la méthode que nous proposons pour l'évaluations des propriétés de distance, plus précisément la distance libre des codes à états finis définis par des codeurs à états finis ; cette famille contient les codes à longueur variable conjoints et les codes arithmétiques conjoints.

Au chapitre 4 nous décrivons les bornes proposées sur la distance libre. Ces bornes seront utilisées dans la recherche de codes.

Au chapitre 5, nous présentons les méthodes proposées pour structurer l'ensemble des codes à longueur variable conjoints pour un ensemble de longueurs de mot de codes fixé et pour structurer l'ensemble des codes arithmétiques conjoints pour les valeurs de paramètres caractéristiques fixées.

Le chapitre 6 présente quelques résultats expérimentaux obtenus et les performances des différents algorithmes proposés avant de conclure et de présenter les perspectives de cette thèse au chapitre 7.

Chapitre 2

Codes linéaires et codes non-linéaires

Ce chapitre rappelle les *propriétés de distance* des codes de canal linéaires tels que les *codes en bloc* et les codes *codes convolutifs*. Il introduit ensuite les propriétés de distance des *codes longueur variable conjoints* et les *codes arithmétiques conjoints*. Par souci de simplicité, nous ne considérons que des codes binaires dans la suite ce rapport.

2.1 Propriétés des codes canal linéaires

Ce paragraphe se focalise sur les codes linéaires binaires.

2.1.1 Codes en bloc

Soit la source X à valeurs dans l'alphabet $\mathcal{A}_M = \{a_1, a_2, \ldots, a_M\}$. A chaque symbole $a_i : 1 \leq i \leq M$, le code en bloc binaire associe un *mot de code* c_i composé de n bits. L'ensemble des mots de code $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ est nommé le *code* ou le *dictionnaire* et n est nommé *longueur du bloc*.

Exemple 2.1 *Soit la source X_4 à valeurs dans l'alphabet $\mathcal{A}_4 = \{00, 01, 10, 11\}$. $\mathcal{C}_4 = \{000, 011, 101, 110\}$ est un dictionnaire possible qu'on peut associer à $\mathcal{A}_4$.*

Definition 2.1 *Le rendement ou taux du code en bloc $\mathcal{C}$, R_c permet de mesurer la redondance introduite et est défini par : $R_c = \frac{\log_2 M}{n}$. Pour le code $\mathcal{C}_4$ de l'exemple 2.1, $R_c = \frac{2}{3}$.*

Definition 2.2 *Le poids de Hamming, $w_H(\boldsymbol{x})$, d'une séquence binaire $\boldsymbol{x}$, est le nombre d'éléments non nuls de $\boldsymbol{x}$. Par exemple, $w_H(000) = 0$, $w_H(100) = 1$, $w_H(101) = 2$ et $w_H(111) = 3$.*

Definition 2.3 *La distance de Hamming d_H entre deux séquences binaires de même longueur $\boldsymbol{x}$, $\boldsymbol{y}$ est le nombre de positions dans lesquelles leurs composants diffèrent, c'est-à-dire le poids de Hamming de leur somme bit à bit modulo 2 : $d_H(\boldsymbol{x}, \boldsymbol{y}) = w_H(\boldsymbol{x} + \boldsymbol{y})$. Par exemple, $d_H(011, 101) = w_H(110) = 2$.*

Definition 2.4 *La distance minimale, $d_{\min}$, d'un ensemble de mots de code $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ est la plus petite distance de Hamming entre toutes les paires de mots de code distincts.*

$$d_{\min} = \min_{\substack{(c_i, c_j) \in \mathcal{C}^2 \\ c_i \neq c_j}} d_H(c_i, c_j) \tag{2.1}$$

Pour le code $\mathcal{C}_4$ de l'exemple 2.1, la distance minimale est $d_{\min} = 2$.

La distance minimale d'un code en bloc détermine sa capacité de détection et de correction d'erreurs. Un code en bloc de distance minimale $d_{\min}$ pourra détecter $d_{\min} - 1$ erreurs et corriger $\left\lfloor \frac{d_{\min}-1}{2} \right\rfloor$ erreurs dans un mot de code. Par conséquent, un code en bloc associé à l'alphabet $\mathcal{A}_M$ peut être représenté par $C(n, M, d_{\min})$, où n est la longueur de bloc, $d_{\min}$ est la distance minimale du code, et M est le nombre de mots de code.

Definition 2.5 *Soit k un nombre entier positif. On dit que le code en bloc $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ avec $M = 2^k$, est linéaire si quel que soit $(c_i, c_j) \in \mathcal{C}^2$ et quel que soit $(\lambda_1, \lambda_2) \in \{0, 1\}^2$, on a : $\lambda_1 \cdot c_i + \lambda_2 \cdot c_j \in \mathcal{C}$. Dans ce cas, le code peut être désigné par $C(n, k, d_{\min})$. Le code $\mathcal{C}_4$ de l'exemple 2.1 est linéaire, alors que le code $\mathcal{C}'_4 = \{111, 011, 101, 110\}$ n'est pas linéaire car $111 + 011 = 100 \notin \mathcal{C}'_4$.*

Si le code $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ est linéaire, alors la distance minimale du code est le plus petit poids de Hamming dans $\mathcal{C}$ différent de 0.

$$d_{\min} = \min_{\substack{c_i \in \mathcal{C} \\ w_{\mathrm{H}}(c_i) \neq 0}} w_{\mathrm{H}}(c_i), \tag{2.2}$$

et son rendement est défini par $R_{\mathrm{c}} = \frac{\log_2 M}{n} = \frac{k}{n}$.

Plusieurs bornes sur la distance minimal du code $\mathcal{C}(n, M, d)$ sont disponibles dans la littérature. Nous rappellerons quelques-unes de ces bornes dans le chapitre 4.

2.1.2 Codes convolutifs

Un *code convolutif* (CC) [Eli55] est généré par un registre à décalage linéaire à état-finis. En général, le registre est composé de L étages de k bits et de n fonctions linéaires. Les données (binaires pour nous) à l'entrée du codeur sont décalées dans et tout au long du registre à décalage par blocs de k bits. Le nombre de bits de sortie pour chaque séquences de k bits en entrée est de n bits. Par conséquent, comme pour le code en bloc de k bits linéaire, le rendement R_c du code convolutif est défini par $R_c = \frac{k}{n}$.

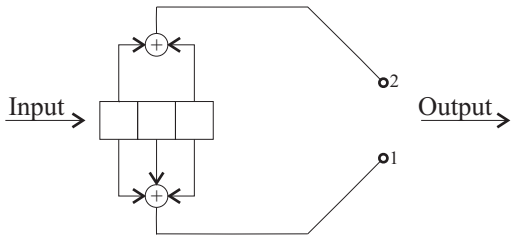


FIGURE 2.1 – Codeur convolutif $k = 1$, $L = 3$ et $n = 2$.

L et $m = L - 1$ sont nommés respectivement la *longueur de contrainte* et la *mémoire* du code convolutif.

Exemple 2.2 *La figure 2.1 montre un exemple de codeur convolutif dans le quel* $k = 1$, $L = 3$, *et* $n = 2$.

A la différence fondamentale des codes en bloc, la sortie d'un codeur convolutif correspondante à un symbole courant dépend non seulement de ce symbole, mais aussi des m symboles précédemment codés (stockés dans le registre).

Dans le cas d'un code convolutif *récursif*, en plus de dépendre des symboles courant et précédemment codés, la sortie courante dépend aussi des sorties précédentes. Par souci de simplicité, nous considérons ici les codes convolutifs *non-récursifs* pour illustrer nos exemples.

2.1.2.1 Représentations graphiques du code convolutif

Il existe plusieurs représentations graphiques d'un codeur convolutif. La première est le *diagramme du codeur* montrée par la figure 2.1.

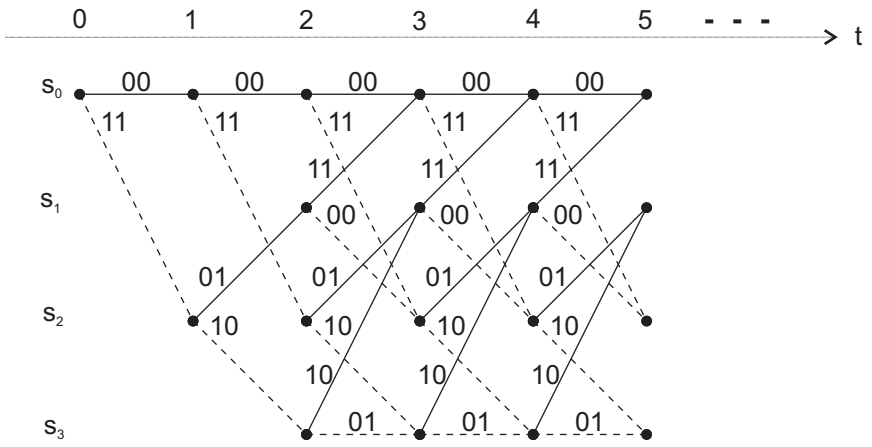


FIGURE 2.2 – Treillis de l'exemple 2.2

Une deuxième représentation nommée *treillis* du code convolutif montre l'évolution temporelle du codeur et est adaptée pour le décodage et l'évaluation des performances de correction d'erreurs. La figure 2.2 représente le treillis associé au codeur de l'exemple 2.2. Sur cette figure, les états s_i, $(i = 0, \cdots, 4)$, représentent les différentes combinaisons possibles des $m \times k = 2$ derniers bits dans le registre à décalage, les pointillés correspondent à une entrée 1 et les traits pleins correspondent à une entrée 0. Les étiquettes sur les transitions correspondent aux symboles de code générés.

Definition 2.6 *On appelle* mot de code convolutif *toute séquence générée par le treillis du code convolutif et qui part de l'état initial s_0 et fini dans le même état s_0. Cette définition sous-entend que les mots de code peuvent être de longueur finies ou (semi-)infinies.*

Definition 2.7 *Le* code convolutif $\mathcal{C}$ *est l'ensemble des mots de code finis ou semi-finis générés par le codeur convolutif.*

Remarque 2.1 *Grâce à l'opération de convolution, le code convolutif $\mathcal{C}$ a une structure linéaire, c'est-à-dire quel que soit $(c_1, c_2) \in \mathcal{C}^2$ de même longueur n (finie ou semi-infinie) et quel que soit $(\lambda_1, \lambda_2) \in \{0, 1\}^2$, la combinaison linéaire $\lambda_1 c_1 + \lambda_2 c_2$ est un mot de code de longueur n appartenant à $\mathcal{C}$.*

On constate que le treillis associé au code convolutif présente une structure totalement répétitive. Cette structure répétitive suggère qu'on peut encore réduire le

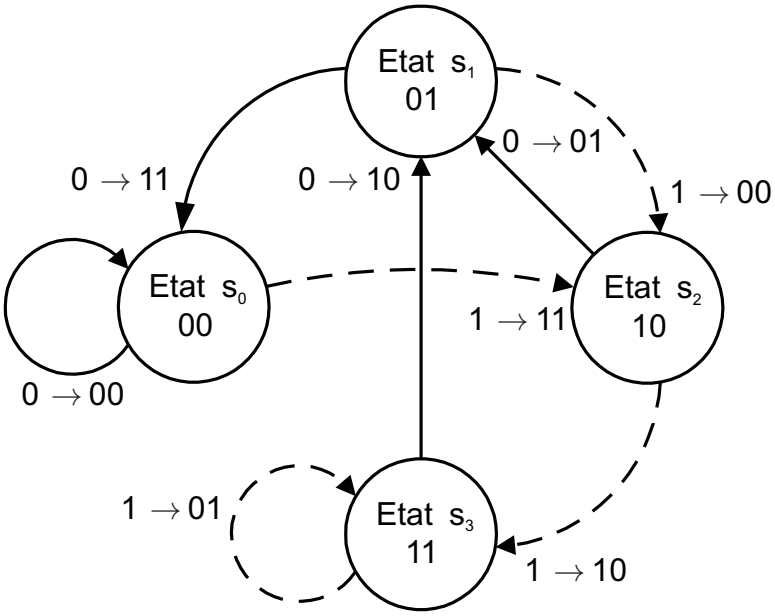


FIGURE 2.3 – Diagramme d'états de l'exemple 2.2

treillis pour obtenir le *diagramme d'états*. La figure 2.3 montre le diagramme d'état obtenu à partir du treillis de la figure 2.2. Cette représentation est la plus compacte du code convolutif et est particulièrement adaptée pour l'évaluation de ses propriétés de distance. Cette représentation ne fait pas apparaitre l'évolution temporelle, et contient $2^{k \cdot m}$ états et 2^k transitions à partir de chaque état. C'est un graphe orienté avec un nombre d'états et de transitions finis, c'est-à-dire une machine à états finis.

2.1.2.2 Propriétés de distance des codes convolutifs

Plusieurs techniques de décodages peuvent être envisagées pour un code convolutif. Parmi ces techniques, on peut citer le décodage par *syndrome* et le décodage par le *maximum de vraisemblance* (MV). Ce dernier est le plus utilisé et plusieurs algorithmes très performants ont été développés (décodage de Viterbi par exemple).

Soit $\boldsymbol{x}$ un mot de code appartenant à $\mathcal{C}$ (où $\mathcal{C}$ est le code convolutif) de longueur η fourni par le codeur convolutif et transmis à un récepteur, et soit $\boldsymbol{y}$ la séquence reçue. La détection au sens du MV dans le domaine code du décodeur convolutif consiste à choisir dans le treillis parmi toutes les séquences de longueur η qui partent de l'état initial s_0 et qui finissent dans le même état s_0 la séquence $\hat{\boldsymbol{x}}$ qui a la plus

grande vraisemblance c'est-à-dire :

$$\hat{\boldsymbol{x}} = \arg\max_{\boldsymbol{x}_d \in \mathcal{C}} p(\boldsymbol{y}|\boldsymbol{x}_d). \tag{2.3}$$

Pour assurer la terminaison dans s_0, toute séquence de symboles codée par le codeur convolutif doit être prolongée par $k \cdot m$ zéros. Si $d_{\mathrm{H}}(\boldsymbol{x}, \hat{\boldsymbol{x}}) = 0$, il n'y a pas d'erreurs commises lors du décodage. Sinon, il y'a eu erreurs de décodage.

Dans le cas d'un *canal binaire symétrique* (CBS), c'est-à-dire un canal qui avec une certaine probabilité donnée inverse un bit $b \in \{0, 1\}$ en $\bar{b} \in \{0, 1\}$ (tel que $b \oplus \bar{b} = 1$), le MV revient à choisir la séquence $\hat{\boldsymbol{x}}$ qui est à la plus petite distance de Hamming de la séquence $\boldsymbol{y}$ reçue. Dans ce cas, pour commettre le moins d'erreurs de décodage, il est important que les distances de Hamming entre les mots de code de même longueur η ($1 \leqslant \eta < \infty$) qui partent de l'état s_0 et qui finissent dans l'état s_0 soient les plus grandes possibles. La plus petite de ces distances de Hamming est la *distance minimale libre* ou *distance libre* (d_{libre}) du code convolutif. Cette distance libre est donc un paramètre très important qui permet de caractériser les performances de détection et correction d'erreurs du code convolutif.

Le code convolutif étant linéaire, il contiendra toujours le mot de tous zéros et donc la distance libre du code correspond au plus petit poids de Hamming non nul des séquences dans le domaine code qui divergent de l'état s_0 et qui re-convergent dans le même état, c'est-à-dire au plus petit poids de Hamming non nul dans l'ensemble des mots de code. Pour trouver d_{libre}, [Vit71] propose d'énumérer le nombre de mots de code avec un certain poids de Hamming d avec une *fonction génératrice* $G(D)$:

$$G(D) = \sum_{d=1}^{\infty} A_d D^d, \tag{2.4}$$

où A_d est le nombre de mots de code de poids de Hamming d et D est une variable symbolique permettant d'indiquer les poids de Hamming. La plus petite valeur de d pour le quelle A_d diffère de 0 correspond à la distance libre du code.

Pour cette fin, [Vit71] propose de redessiner le diagramme d'état du code convolutif en *graphe de fluence* avec un signal d'entrée $s_{\mathrm{in}} = s_0$ et un signal de sortie $s_{\mathrm{out}} = s_0$. Sur ce nouveau graphe les étiquettes sur les transitions représentent les poids de Hamming des transitions. La figure 2.4 montre le graphe de fluence obtenu pour l'exemple 2.2.

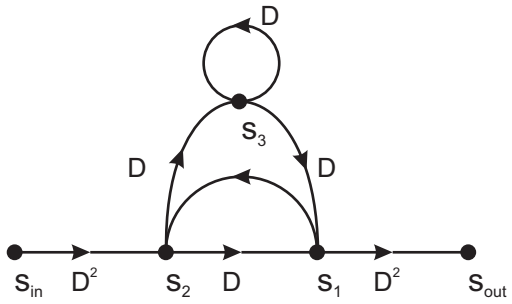


FIGURE 2.4 – Graphe de fluence de l'exemple 2.2 étiqueté avec les poids de Hamming

La fonction génératrice $G(D)$ est obtenue en calculant la fonction de transfert entre s_{in} et s_{out}

$$G(D) = \frac{s_{\text{out}}}{s_{\text{in}}}. \tag{2.5}$$

En appliquant les règles d'évaluation pour les graphes de fluence sur la figure 2.4 on obtient

$$\begin{aligned} s_1 &= Ds_2 + Ds_3 \\ s_2 &= D^2 s_{\text{in}} + s_1 \\ s_3 &= Ds_2 + Ds_3 \\ s_{\text{out}} &= D^2 s_1, \end{aligned}$$

ce qui conduit à

$$G(D) = \frac{s_{\text{out}}}{s_{\text{in}}} = \frac{D^5}{1-2D} = D^5 + 2D^6 + 4D^7 + \cdots .$$

$$G(D) = \sum_{d=5}^{\infty} 2^{d-5} D^d. \tag{2.6}$$

On constate que le premier coefficient différent de 0 est A_5, donc la *distance libre* du code de l'exemple 2.2 est $d_{\text{libre}} = 5$. Dans un sens plus large, la fonction $G(D)$ représente le spectre de distance du code, c'est-à-dire le nombre de chemin à une distance d d'un chemin donné dans le treillis. $G(D)$ permet une évaluation plus fine des performances de correction d'erreur que la connaissance de la seule distance libre. En effet le spectre de distance permet de calculer la *borne de l'union des évènements d'erreurs* qui borne la probabilité P_e qu'un mot de code soit décodé de manière erronée. On a :

$$P_e \leqslant \sum_{d=d_{\text{libre}}}^{\infty} A_d P_d, \tag{2.7}$$

où P_d représente la probabilité qu'un mot de code donné soit confondu avec un autre mot de code qui se trouve à la distance de Hamming d; voir [Vit71] pour plus de détails. Par exemple pour un canal binaire symétrique de probabilité d'erreur p, on a :

$$p_d = \begin{cases} \displaystyle\sum_{e=\frac{d+1}{2}}^{d} \begin{pmatrix} d \\ e \end{pmatrix} p^e (1-p)^{d-e}, & \text{si } d \text{ est impaire} \\ \frac{1}{2} \begin{pmatrix} d \\ \frac{d}{2} \end{pmatrix} p^{\frac{d}{2}} (1-p)^{\frac{d}{2}} + \displaystyle\sum_{e=\frac{d}{2}+1}^{d} \begin{pmatrix} d \\ e \end{pmatrix} p^e (1-p)^{d-e}, & \text{si } d \text{ est paire} \end{cases} \tag{2.8}$$

[Ast86] utilise une variante de l'algorithme de Dijkstra sur le graphe de fluence de la figure 2.4 pour calculer directement la distance libre sans générer le spectre de distance. En effet, en étiquetant chaque transition du graphe par son poids de Hamming, on constate que trouver la distance libre revient à trouver le chemin entre s_{in} et s_{out} avec le plus petit poids cumulé. Puisque toutes les transitions sont de poids positif ou nul, ce problème peut être résolu par l'algorithme de Dijkstra [GM84].

Un certain nombre de bornes sur la distance libre des codes convolutifs existent dans la littérature. Nous en rappellerons quelques-unes dans le chapitre 5.

2.2 Codes conjoints (non-linéaires)

Un codeur de source génère des séquences de longueur variable pour obtenir de la compression. Un code à longueur variable (CLV) de type Huffman génère le code symbole par symbole (en substituant chaque symbole par une séquence à longueur variable). Un code arithmétique génère le mot de code en ligne (voir paragraphe 2.2.2). Ces deux types de code peuvent inclure de la redondance et donc devenir des codes conjoints. Dans ce cas, le décodage sera lui aussi (forcement) conjoint. A remarquer qu'un décodage conjoint est possible (mais pas nécessaire) aussi pour un système tandem

2.2.1 Code à longueurs variables, sans et avec redondance

Considérons une source X avec l'alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$ et les probabilités associées $\boldsymbol{p} = (p_1, p_2, \ldots, p_M)$. Les *Codes à Longueurs Variables* (CLVs) tel

que les codes d'Huffman [Huf52] associent à chaque symbole a_i un mot de code $c_i \in \{0,1\}^{\ell_i}$ dans un ensemble de mots de code $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$. La longueur en bits de c_i est ℓ_i, $i = 1, \ldots, M$. L'ensemble des mots de code $\mathcal{C}$ est appelé aussi *dictionnaire*. Le $j^{\text{ième}}$ bit de c_i est nommé c_i^j. L'ensemble des longueurs $\ell_i : 1 \leqslant i \leqslant M$ est décrit par le *vecteur de longueurs* $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$.

Exemple 2.3 *Soit la source X_4 à valeurs dans $\mathcal{A}_4 = \{a, b, c, d\}$ avec $\boldsymbol{p}_4 = \left\{\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8}\right\}$. Les codes $\mathcal{C}_4^0 = \{1, 01, 000, 001\}$ ($\boldsymbol{\ell}_4^0 = (1,2,3,3)$), $\mathcal{C}_4^1 = \{1, 00, 000, 001\}$ ($\boldsymbol{\ell}_4^1 = (1,2,3,3)$) et $\mathcal{C}_4^2 = \{1, 01, 11, 001\}$ ($\boldsymbol{\ell}_4^2 = (1,2,2,3)$) sont différents dictionnaires qu'on peut associer à X_4.*

Definition 2.8 *On dit que le dictionnaire $\mathcal{C}_M$ est* préfixe *si aucun de ses mots de code n'est le préfixe d'un autre mot de code. Pour qu'un code soit préfixe, il faut que les longueurs des mots de code $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ satisfassent l'inégalité de Kraft [CT91, Pro95] :*

$$\sum_{i=1}^{M} 2^{-\ell_i} \leqslant 1. \tag{2.9}$$

Inversement, pour tout jeu de longueurs ℓ qui satisfait l'inégalité de Kraft, il existe un code préfixe avec ces longueurs.

Pour un vecteur de longueurs $\boldsymbol{\ell}$ donné, on définit :

$$\text{Kraft}(\boldsymbol{\ell}) = \sum_{i=1}^{M} 2^{-\ell_i}. \tag{2.10}$$

Pour l'exemple 2.3, on a

- $\mathcal{C}_4^0$ est préfixe et $\text{Kraft}(\boldsymbol{\ell}_4^0) = 1$,
- $\mathcal{C}_4^1$ n'est pas préfixe car c_2 est préfixe de c_3 et de c_4, $\text{Kraft}(\boldsymbol{\ell}_4^1) = 1$,
- $\mathcal{C}_4^2$ n'est pas préfixe car c_1 est préfixe de c_3 et $\text{Kraft}(\boldsymbol{\ell}_4^2) = 1,125 > 1$.

Dorénavant, on considère que (2.9) est satisfaite et on nomme $\boldsymbol{\ell}$ un *Vecteur de Kraft.*

Definition 2.9 *L'entropie $H(X)$ d'une source X représente la quantité moyenne d'information associée à chaque symbole généré par la source [Sha48]. Elle est définie comme :*

$$H = -\sum_{i=1}^{M} p_i \log_2(p_i) \text{ bits/symbole}. \tag{2.11}$$

Pour la source X_4 de l'exemple 2.3, l'entropie

$$H_4 = \frac{1}{2}\log_2(2) + \frac{1}{4}\log_2(4) + \frac{1}{8}\log_2(8) + \frac{1}{8}\log_2(8) = 1,75 \text{ bits/symbole} \quad (2.12)$$

Definition 2.10 *La* longueur moyenne du code *(ℓ_{moy}) est le nombre moyen de bits utilisés pour représenter chaque symbole de la source :*

$$\ell_{\text{moy}} = \sum_{i=1}^{M} p_i \ell_i \text{ bits/symbole.} \quad (2.13)$$

Théorème 2.2 *Théorème de Shannon pour le codage de source [Sha48]*
Soit une source X d'entropie H. Quel que soit $\epsilon > 0$, il existe un code de longueur moyenne $H \leqslant \ell_{\text{moy}} < H + \epsilon$ qui permet de coder la source X de façon réversible (sans pertes).

Definition 2.11 *Soit X une source d'entropie H, et $\mathcal{C}$ un CLV associé à X de longueur moyenne ℓ_{moy}. La* redondance résiduelle*, ρ_c, définie dans [CT91], est la différence entre la longueur moyenne ℓ_{moy} et l'entropie H.*

$$\rho_c = \ell_{\text{moy}} - H \text{ bits/symbole} \quad (2.14)$$

La redondance permet de mesurer l'efficacité en termes de compression. Si $\rho_c = 0$, alors le code est sans redondance et (2.9) doit être une égalité. Si $\rho_c > 0$, le CLV a introduit de la redondance, ce qui peut lui conférer une certaine *capacité de détection et de correction d'erreurs.*
Pour l'exemple 2.3, si on associe le code :

- $\mathcal{C}_4^0 = \{1, 01, 000, 001\}$, on a $\ell_{\text{moy}} = 1,75$ bits/symbole et $H_4 = 1,75$ bits/symbole, donc $\rho_c = 0$ bits/symboles, par conséquent pas de redondance,
- $\mathcal{C}_4^4 = \{1, 000, 011, 001\}$, on a $\ell_{\text{moy}} = 2$ bits/symboles et $H_4 = 1,75$ bits/symbole, donc $\rho_c = 0,25$ bits/symboles, par conséquent le code introduit de la redondance.

Le *décodage source-canal conjoint* exploite la redondance introduite par le codeur de source pour corriger d'éventuelles erreurs de transmission.

L'idée du *CLV conjoint* est d'introduire de la redondance pour construire des dictionnaires robustes à l'égard d'erreurs de transmission. Le CLV conjoint s'utilise uniquement avec un décodeur conjoint. Comme pour les codes de canal linéaires, les

performances de corrections d'erreurs des CLV conjoints sont caractérisées d'abord par la *distance libre* (une caractérisation plus fine est possible à travers le *spectre de distance* [Vit71, VO79]). Les CLVs conjoints étant non linéaires, donc la caractérisation de ces paramètres (distance libre et spectre de distance) est peu plus difficile. Nous verrons dans le paragraphe 2.2.3 comment définir ces paramètres pour les CLV conjoints.

2.2.2 Code arithmétique, sans et avec redondance

2.2.2.1 Codage arithmétique de base

Soit X une source à valeurs dans $\mathcal{A} = \{a_1, a_2, \ldots, a_M\}$ avec les probabilités d'occurrence $\boldsymbol{p} = \{p_1, p_2, \ldots, p_M\}$. L'idée de base du *Codage Arithmétique* (CA) est d'assigner à chaque séquence de symboles source, $\boldsymbol{s}_N \in \mathcal{A}^N : 1 \leqslant N < \infty$ un sous-intervalle unique $I_{\boldsymbol{s}}$ de l'intervalle unité $[0, 1)$ obtenu en découpant itérativement l'intervalle unité $[0, 1)$. Tout nombre rationnel $c_{\boldsymbol{s}} \in I_{\boldsymbol{s}}$ sera un mot de code pouvant décrire $\boldsymbol{s}_N$ sans ambiguïté si N est connu.

Soit $[l, h)$ l'intervalle de code à la fin d'une certaine itération du codage, et $w = h - l$ la largeur de l'intervalle. Au début (à la première itération) $l = 0$ et $h = 1$. Le codeur arithmétique partitionne $[l, h)$ en M sous-intervalles $[F^0_{i-1}, F^0_i)$ de largeur $w^0_i = w \times p_i$ pour $1 \leqslant i \leqslant M$. $F^0_0 = l$ et $F^0_M = h$ et le sous-intervalle $[F^0_{i-1}, F^0_i)$ est associé au symbole a_i. Si le premier symbole de la séquence $\boldsymbol{s}_N$ à coder est a_ν, alors on choisit $[l, h) = [F^0_{\nu-1}, F^0_\nu)$ qu'on partitionne à son tour en M sous-intervalles $[F^1_{i-1}, F^1_i)$ de largeur $w^1_i = w \times p_i$ pour $1 \leqslant i \leqslant M$. Si le deuxième symbole de la séquence $\boldsymbol{s}_N$ est a_λ, alors, le sous-intervalle $[F^1_{\lambda-1}, F^1_\lambda)$ est choisi et ainsi de suite. Si a_μ est le dernier symbole à coder, on choisit l'intervalle $[l, h) = [F^{N-1}_{\mu-1}, F^{N-1}_\mu)$ comme intervalle final de codage de $\boldsymbol{s}_N$.

Exemple 2.4 *Soit la source X à valeurs dans l'alphabet $\mathcal{A} = \{a_1, a_2, a_3\}$ avec $\boldsymbol{p} = (0.7, 0.1, 0.2)$. Soit la séquence $\boldsymbol{s}_4 = (a_1 a_3 a_1 a_2)$ à coder. Initialement $[l, h) = [0, 1)$. A la première itération, les sous-intervalles associés à chaque symbole sont $[F^0_0, F^0_1) = [0, 0.7)$, $[F^0_1, F^0_2) = [0.7, 0.8)$ et $[F^0_2, F^0_3) = [0.8, 1)$ voir figure 2.5 (a). Le premier symbole de $\boldsymbol{s}_4$ est a_1 donc le sous-intervalle $[l, h) = [0, 0.7)$ est choisi puis subdivisé. A la deuxième itération, on obtient les sous-intervalles suivants : $[F^1_0, F^1_1) = [0, 0.49)$, $[F^1_1, F^1_2) = [0.49, 0.56)$ et $[F^1_2, F^1_3) = [0.56, 0.7)$ (figure 2.5 (b)). Le deuxième symbole*

de $\boldsymbol{s}_4$ *est* a_3 *donc le sous-intervalle* $[l, h) = [0.56, 0.7)$ *est choisi. A la troisième itération les sous-intervalles sont* $[F_0^2, F_1^2) = [0.56, 0.658)$, $[F_1^2, F_2^2) = [0.658, 0.672)$ *et* $[F_2^2, F_3^2) = [0.672, 0.7)$ *figure 2.5 (c)). Le troisième symbole à coder est* a_1, *le sous-intervalle* $[l, h) = [0.56, 0.658)$ *est choisi. Ensuite, on obtient les sous-intervalles* $[F_0^3, F_1^3) = [0.56, 0.6286$, $[F_1^3, F_2^3) = [0.6286, 0.6384)$ *et* $[F_2^2, F_3^2) = [0.6384, 0.658)$ *(figure 2.5 (d)). le dernier symbole à coder est* a_2 *donc l'intervalle final de codage est* $[l, h) = [0.6286, 0.6384)$.

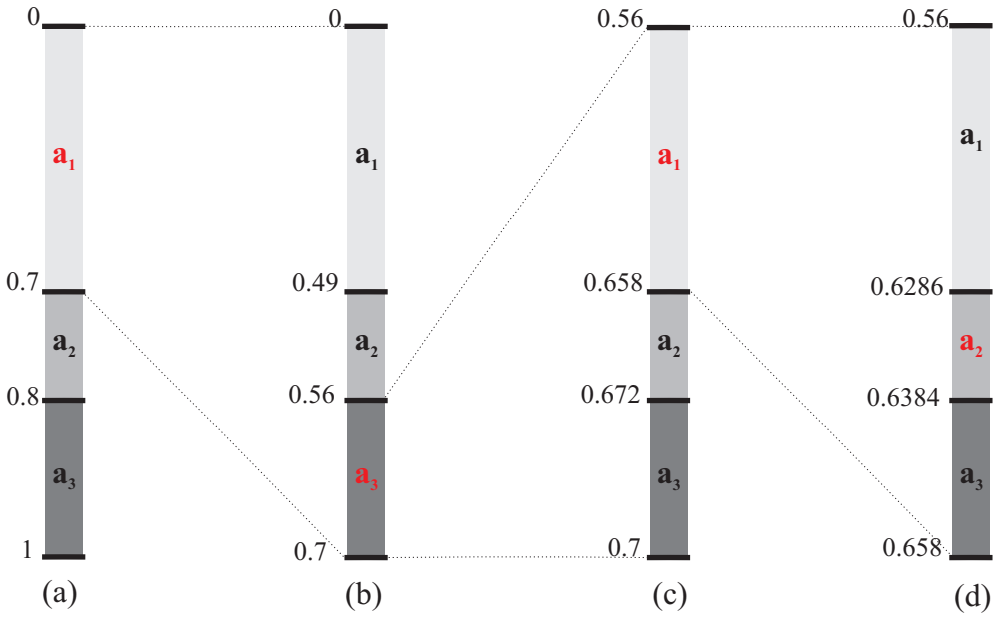


FIGURE 2.5 – Codage de la séquence $\boldsymbol{s}_4$ de l'exemple 2.4

Par construction, la largeur de l'intervalle final de codage $w = h - l$ correspond à la probabilité d'occurrence de la séquence $\boldsymbol{s}_N$, $P_{\boldsymbol{s}}$, obtenue par le produit des probabilités des différents symboles qui la composent. Pour la séquence $\boldsymbol{s}_N$ de l'exemple 2.4,

$$\begin{aligned} P_{\boldsymbol{s}} &= p_1 \times p_3 \times p_1 \times p_2 = 0.7 \times 0.2 \times 0.7 \times 0.1 = 0.0098 \\ w &= h - l = 0.6384 - 0.6286 = 0.0098. \end{aligned}$$

Une fois l'intervalle final de codage trouvé, on choisit un nombre $T_{\boldsymbol{s}}$ dans cet intervalle et sa représentation binaire est associée à la séquence $\boldsymbol{s}_N$. Pour l'exemple 2.4, on peut choisir

$$T_{\boldsymbol{s}} = \frac{l + h}{2} = \frac{0.6286 + 0.6384}{2} = 0.6335, \tag{2.15}$$

dont la représentation binaire est $T_{\boldsymbol{s}} = 0.10100010\ldots$

On constate que cette représentation binaire peut être de longueur infinie. Ce qui n'est pas idéal pour une application pratique. Donc il est nécessaire de tronquer cette représentation binaire sur une longueur $\ell_{\boldsymbol{s}}$ finie pour obtenir un nombre $\bar{T}_{\boldsymbol{s}}$ en prenant soins que ce dernier nombre reste toujours dans l'intervalle final de codage $[l, h)$. Cette condition peut être assurée en choisissant $T_{\boldsymbol{s}}$ comme le milieu de l'intervalle final de codage et en tronquant sa représentation binaire à $\ell_{\boldsymbol{s}}$ bits avec :

$$\ell_{\boldsymbol{s}} = \left\lceil \log_2\left(\frac{1}{P_{\boldsymbol{s}}}\right)\right\rceil + 1, \tag{2.16}$$

Preuve: Il suffit de montrer que $T_{\boldsymbol{s}} - \bar{T}_{\boldsymbol{s}} \leqslant \frac{P_{\boldsymbol{s}}}{2}$. Soit $(0.b_1b_2\cdots b_{\ell_{\boldsymbol{s}}}b_{\ell_{\boldsymbol{s}}+1}\cdots)_2 : b_i \in \{0,1\}$, la représentation binaire de $T_{\boldsymbol{s}}$. Dans ce cas $\bar{T}_{\boldsymbol{s}}$ aura pour représentation binaire $(0.b_1b_2\cdots b_{\ell_{\boldsymbol{s}}})_2$.

$$T_{\boldsymbol{s}} - \bar{T}_{\boldsymbol{s}} \leqslant (0.0\cdots 0 b_{\ell_{\boldsymbol{s}}+1}b_{\ell_{\boldsymbol{s}}+2}\cdots)_2 \leqslant 2^{-\ell_{\boldsymbol{s}}} \tag{2.17}$$

En remplaçant dans (2.17) $\ell_{\boldsymbol{s}}$ par sa valeur fournie par (2.16) on obtient

$$T_{\boldsymbol{s}} - \bar{T}_{\boldsymbol{s}} \leqslant 2^{-\left(\left\lceil \log_2\left(\frac{1}{P_{\boldsymbol{s}}}\right)\right\rceil + 1\right)} \leqslant 2^{-\left(\log_2\left(\frac{1}{P_{\boldsymbol{s}}}\right)+1\right)}, \tag{2.18}$$

qui conduit à $T_{\boldsymbol{s}} - \bar{T}_{\boldsymbol{s}} \leqslant \frac{P_{\boldsymbol{s}}}{2}$. □

Pour l'exemple 2.4, on a $\ell_{\boldsymbol{s}} = 8$, qui conduit à une représentation binaire de $\bar{T}_{\boldsymbol{s}}$ de (0.10100010) et $\bar{T}_{\boldsymbol{s}} = 0.6328125 \in [0.6286, 0.6384)$.

2.2.2.2 Performances du codage arithmétique

Comme pour les CLVs les performances de compression du codeur arithmétique sont évaluées avec la redondance résiduelle $\rho_{\mathrm{c}} = \ell_{\mathrm{moy}} - H$ définie par (2.14) à la page 40. Rappelons que le codage arithmétique de base code les symboles par séquence de symboles $\boldsymbol{s}_N \in \mathcal{A}_M^N : 1 \leqslant N < \infty$. Donc pour comparer ℓ_{moy} à H, nous allons d'abord estimer l'entropie H^N et la longueur moyenne ℓ_{moy}^N pour les séquences de longueur N puis en déduire H et ℓ_{moy}.

$$H^N = \sum_{\boldsymbol{s}_N} P_{\boldsymbol{s}} \log_2\left(\frac{1}{P_{\boldsymbol{s}}}\right) \tag{2.19}$$

$$\ell_{\mathrm{moy}}^N = \sum_{\boldsymbol{s}_N} P_{\boldsymbol{s}}\ell_{\boldsymbol{s}} \tag{2.20}$$

En remplaçant dans (2.20) $\ell_{\boldsymbol{s}}$ par sa valeur donnée par (2.16) on obtient

$$\ell_{\text{moy}}^{N} = \sum_{\boldsymbol{s}_N} P_{\boldsymbol{s}} \times \left(\left\lceil \log_2 \left(\frac{1}{P_{\boldsymbol{s}}} \right) \right\rceil + 1 \right) \tag{2.21}$$

$$\sum_{\boldsymbol{s}_N} P_{\boldsymbol{s}} \times \log_2 \left(\frac{1}{P_{\boldsymbol{s}}} \right) \leqslant \ell_{\text{moy}}^{N} < \sum_{\boldsymbol{s}_N} P_{\boldsymbol{s}} \times \left(\log_2 \left(\frac{1}{P_{\boldsymbol{s}}} \right) + 2 \right) \tag{2.22}$$

$$H^N \leqslant \ell_{\text{moy}}^{N} < H^N + 2. \tag{2.23}$$

Or, on a pour une source sans mémoire

$$H = \frac{H^N}{N}, \tag{2.24}$$

$$\ell_{\text{moy}} = \frac{\ell_{\text{moy}}^{N}}{N}, \tag{2.25}$$

et par conséquent en divisant (2.23) par N, on obtient :

$$H \leqslant \ell_{\text{moy}} < H + \frac{2}{N}. \tag{2.26}$$

Pour des longueurs de séquences N très grandes, la longueur moyenne tend vers l'entropie de la source.

Au niveau du codeur, on constate que le train binaire n'est transmis que lorsque toute la séquence de données est codée. Si la séquence de données est très longue, cela pose des problèmes pour les applications qui ont des contraintes de délais d'acheminement et de décodage. De plus la taille de l'intervalle final de codage tend vers zéro, ce qui pourrait ramener à utiliser un très grand nombre de bit pour représenter l'intervalle final (même après la troncature voir (2.16)). Pour résoudre ce problème, le codeur arithmétique a été adapté de manière à pouvoir travailler en précision finie.

2.2.2.3 Implémentation pratique du CA

Le codage arithmétique de base consiste à attribuer à toute séquence de données un unique intervalle compris entre $[0, 1)$, ensuite choisir un nombre dans cet intervalle dont la représentation binaire sera associée à la séquence codée (voir le paragraphe 2.2.2.1).

Lorsque le codage de la séquence de données commence, si le codage d'un certain nombre de symboles conduit à un intervalle qui se trouve entièrement dans le demi intervalle $[0, \frac{1}{2})$, l'intervalle final de codage sera lui aussi entièrement situé dans le

demi intervalle $[0, \frac{1}{2})$. Or quel que soit le nombre réel appartenant à l'intervalle $[0, \frac{1}{2})$, son bit de poids fort est le bit 0. Donc la représentation binaire finale aura pour bit de poids fort le bit 0. De même, si le codage d'un certain nombre de symboles conduit dans le demi intervalle $[\frac{1}{2}, 1)$, alors l'intervalle final de codage sera lui aussi entièrement confiné dans l'intervalle $[\frac{1}{2}, 1)$ et pour les mêmes raisons, la représentation binaire finale aura pour bit de poids fort le bit 1. Donc dans un cas ou dans l'autre, le bit de poids fort peut être envoyé sans attendre la fin du codage complet de la séquence.

En transmettant le bit de poids fort, le reste de la séquence binaire est décalé vers la gauche, ce qui revient à élargir l'intervalle courant en multipliant l et h par deux. Par conséquent, la largeur de l'intervalle reste toujours supérieure à $\frac{1}{4}$, ce qui permet d'éviter des problèmes de précisions de calcul.

Dans le cas où l'intervalle est inclus dans l'intervalle $[\frac{1}{2}, 1)$, 1 est soustrait des valeurs obtenues après extension pour ne pas qu'elles dépassent 1.

Par contre, si le codage du symbole conduit à un intervalle $[l, h)$ qui se trouve dans l'intervalle $[\frac{1}{4}, \frac{3}{4})$, alors l'intervalle chevauche la valeur $\frac{1}{2}$, c'est-à-dire que la valeur du nombre qui représente l'intervalle peut être supérieure ou inférieure à $\frac{1}{2}$. Mais à ce stade, cette valeur est inconnue. Dans le cas où la valeur est inférieure à $\frac{1}{2}$, elle est comprise entre $[\frac{1}{4}, \frac{1}{2})$, donc sa représentation binaire est sous forme de $0,01xxxx\cdots$. Dans le cas où elle est supérieure à $\frac{1}{2}$, elle est comprise entre $[\frac{1}{2}, \frac{1}{4})$ est sa représentation binaire est sous forme de $0,10xxx\cdots$. Il est à constater que le premier bit et le deuxième bit sont opposés, mais à ce stade, étant donné que la valeur n'est pas connue, aucun bit n'est émis. Par contre, la trace du deuxième bit est gardée en mémoire en incrémentant une variable $follow$ (f). Une mise à jour de l'intervalle est effectuée et $\frac{1}{2}$ est soustrait aux valeurs de l et h pour éviter que celles ci ne dépassent 1. Une fois qu'on revient dans l'un des demi intervalles $[0, \frac{1}{2})$ ou $[\frac{1}{2}, 1)$, le bit de poids fort correspondant est émis suivi de follow bit opposés.

On répète le processus pour tous les symboles suivants, ainsi, le train binaire est généré au fur et à mesure du codage de la séquence.

On constate que la valeur de follow peut augmenter à l'infini, donc en pratique, cette valeur est limitée à $f_{\max}$. Ici, nous utilisons l'approche de [BJWK08], c'est-à-dire, si $f = f_{\max}$ et le symbole courant à coder est tel que la valeur de f peut encore être incrémentée, les probabilités d'occurrence des symboles sont temporairement

changées pour obliger une transmission de bits (c'est-à-dire tel que aucun sous-intervalle de $[l, h)$ correspondant à un symbole ne chevauche $\frac{1}{2}$) .

L'algorithme est décrit dans le tableau 2.1 ([WNC87]) avec les définitions suivantes : FIRST_QTR $= \frac{1}{4}$, HALF $= \frac{1}{2}$ et THIRD_QTR $= \frac{3}{4}$.

Exemple 2.5 *On reprend l'exemple 2.4 où la séquence à coder est* $\boldsymbol{s}_N = (a_1 a_3 a_1 a_2)$. *Soit* $\boldsymbol{b}$ *la séquence binaire à générer.*

- Étape 1, *initialisation :* $[l, h) = [0, 1)$, $f = 0$ *et* $\boldsymbol{b} = (0.)$,
- Étape 2, *codage du premier symbole* a_1

$$\begin{aligned} w &= 1 - 0 = 1, \\ l &= l + w \times p_0 = 0, \\ h &= h + w \times p_1 = 0.7, \end{aligned}$$

- Étape 3, *génération de bit et normalisation : On constate que l'intervalle courant*

  $[0, 0.7) \notin [0, \text{HALF})$,$[0, 0.7) \notin [\text{HALF}, 1)$ *et* $[0, 0.7) \notin$ $[\text{FIRST\_QTR}, \text{THIRD\_QTR})$. *Donc on revient à l'étape 2,*
- *codage du deuxième symbole* a_3 *conduit à* $[l, h) = [0.56, 0.7)$. *A l'étape 3, on constate que* $[0.56, 0.7) \in [\text{HALF}, 1)$. *Donc on émet un bit* 1 $(f = 0)$ *et* $\boldsymbol{b} = (0.1)$. *Ensuite on normalise, on obtient*

  $$l = 2 \times l - 1 = 0.12 \text{ et } h = 2 \times h - 1 = 0.4 \text{ et } f = 0,$$

  $[0.12, 0.24) \in [0, \text{HALF})$ *on émet un bit* 0, $\boldsymbol{b} = (0.10)$, *après normalisation on obtient* $[l, h) = [0.24, 0.8)$
- *codage du troisième symbole* a_1 *conduit à* $[l, h) = [0.24, 0.632)$. *Pas d'émission de bit et pas de normalisation,*
- *codage du dernier symbole* a_2, *on obtient* $[l, h) = [0.5144, 0.5536)$. *Émission d'un bit* 1 $(f = 0)$, $\boldsymbol{b} = (0.101)$. *On normalise ,* $[l, h) = [0.0288, 0.1072)$, *émission d'un bit* 0, $\boldsymbol{b} = (0.1010)$, *on normalise,* $[l, h) = [0.0576, 0.2144)$, *émission d'un bit* 0, $\boldsymbol{b} = (0.10100)$. *On normalise* $[l, h) = [0.1152, 0.4144)$, *émission d'un* 0, $\boldsymbol{b} = (0.101000)$, *qui conduit à* $[l, h) = [0.2304, 0.8288)$. *Ensuite on va à l'étape 2 qui conduit à l'étape 4.*
- Étape 4, $l = 0.2304 <$ FIRST_QTR, *donc émission d'un* 0 *suivit de* $f + 1 = 1$ *bits à 1. qui conduit à* $\boldsymbol{b} = (0.10100001)$.

TABLE 2.1 – Algorithme de codage arithmétique

Algorithme de l'implémentation pratique du codeur arithmétique

1	Initialisation de l'intervalle courant $[l, h)$ et d'une variable *follow* (f)
	$[l, h) = [0, 1)$ et $f = 0$
2	Codage du symbole courant
	- Si a_k est le symbole courant à coder, calculer l'intervalle courant
	$w = h - l \quad , \quad l = l + w \times \sum_{i=1}^{k-1} p_i \quad , \quad h = l + w \times p_k$ avec $p_0 = 0$
	$k = k + 1$
	- S'il n y'a plus de symbole à coder, passer à l'étape 4
3	Émission de bits et mise à jour
	- Si l'intervalle courant $[l, h) \subseteq [0, \text{HALF})$
	émission d'un bit 0 suivi de f bits 1
	$l = 2 \times l \quad , \quad h = 2 \times h \quad , \quad f = 0$
	reprendre la procédure au début de l'étape 3
	- Si l'intervalle courant $[l, h) \subseteq [\text{HALF}, 1)$
	émission d'un bit 1 suivi de f bits 0
	$l = 2 \times l - 1 \quad , \quad h = 2 \times h - 1 \quad , \quad f = 0$
	reprendre la procédure au début de l'étape 3
	- Si l'intervalle courant $[l, h) \subseteq [\text{FIRST\_QTR}, \text{THIRD\_QTR})$
	aucun bit n'est émis
	$l = 2 \times l - \frac{1}{2} \quad , \quad h = 2 \times h - \frac{1}{2} \quad , \quad f = f + 1$
	reprendre la procédure au début de l'étape 3
	- Sinon
	aucun bit n'est émis
	aller à l'étape 2
4	Il n'y'a plus de symbole à coder
	- si $l \leqslant$ FIRST_QTR
	émission d'un bit 0 suivi de f +1 bits 1
	- sinon
	émission d'un bit 1 suivi de f +1 bits 0

La valeur obtenue par la séquence binaire $\boldsymbol{b}$ *est de* $\bar{T}_A^1 = 0.62890625 \in [0.6286, 0.6384)$ *obtenu précédemment.*

2.2.2.4 Codes arithmétiques conjoints

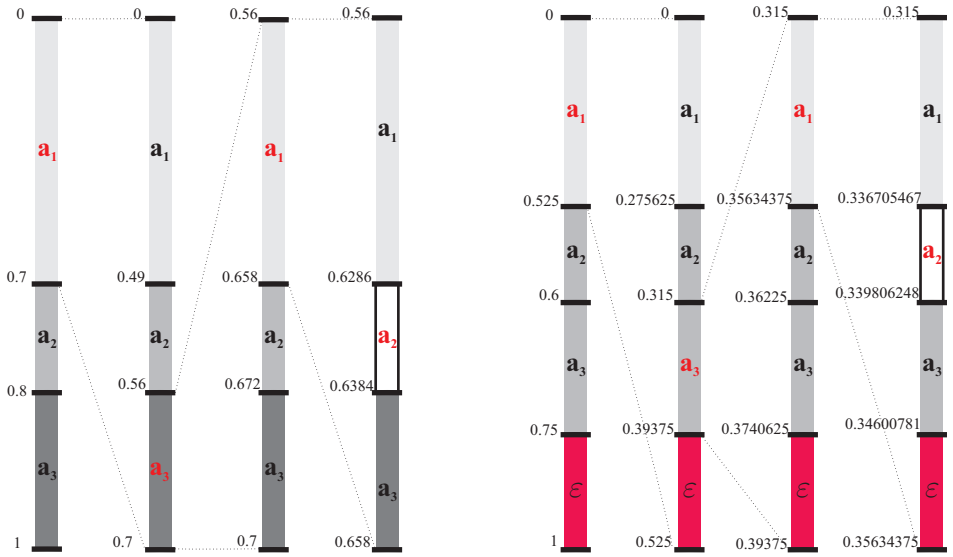
Dans un code arithmétique, il y a très peu de redondance et la sensibilité à l'égard d'erreurs de transmission affectant le train binaire généré est grande. Une modification du schéma de codage par introduction de redondance permet d'obtenir un code arithmétique (CA) conjoint potentiellement plus robuste à l'égard d'erreurs de transmission.

[BCI+97] montre qu'on peut augmenter les performances de détections d'erreurs des CAs conjoints en y introduisant volontairement plus de redondance par l'intermédiaire d'un *symbole interdit* (SI). Ce SI noté ε n'est jamais émis par la source mais une probabilité non nulle p_ε lui est allouée lors de la subdivision de l'intervalle courant $[l, h)$ durant le processus de codage. Par conséquent, les probabilités d'occurrences des symboles lors du processus de codage deviennent $p_i^c = (1 - p_\varepsilon) \times p_i$: $i = 1, \cdots, M$.

La figure 2.6 (a) montre le codage arithmétique de la séquence $\boldsymbol{s}_4$ de l'exemple 2.4 sans introduire de SI, alors que la figure 2.6 (b) montre le codage de la même séquence avec un SI de probabilité $p_\varepsilon = 0.25$. Sans l'introduction du SI, le codage conduit à un intervalle final $[l, h) = [0.6286, 0.63)$ avec une longueur de la séquence binaire générée $\ell_{\boldsymbol{s}} = 8$, voir l'exemple 2.4 (paragraphe 2.2.2.1, page 42). Avec l'introduction du SI, l'intervalle final de codage devient $[l^{SI}, h^{SI}) = [0.336705467, 0.339806248)$ conduisant à $w^{SI} = 0.003100781$ et une longueur de la séquence binaire à générer de $\ell_{\boldsymbol{s}}^{SI} = 10$ bits (voir (2.16) page 43 pour le calcule de $\ell_{\boldsymbol{s}}$). On constate que l'introduction du symbole interdit a ajouté de la redondance supplémentaire ρ^{SI} dans la séquence binaire générée, et cette redondance a été évaluée par [CR00] à

$$\rho^{SI} = -\log_2\left(\frac{1}{(1 - p_\varepsilon)}\right) \text{ bits/symbole.} \tag{2.27}$$

Pour encore améliorer les performances de corrections d'erreurs sans introduire plus de redondance, [Say99] introduit les SI *multiples* (SIM), c'est-à-dire introduire jusqu'à $M + 1$ SIs, $\{\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_M, \varepsilon_{M+1}\}$, en associant à chaque $\varepsilon_i : 1 \leqslant i \leqslant M + 1$



(a) Codage de la séquence sans SI

(b) Codage de la séquence avec SI

FIGURE 2.6 – (a) Codage arithmétique de la séquence $\boldsymbol{s}_4$ de l'exemple 2.4 sans, et (b) avec un SI de probabilité $p_\varepsilon = 0.25$.

une probabilité p_{ε_i} tel que :

$$\sum_{i=1}^{M+1} p_{\varepsilon_i} = p_\varepsilon. \tag{2.28}$$

Avec le SI multiple, la subdivision de l'intervalle courant de codage $[l, h)$ dans le cas le plus général est présentée sur la figure 2.7.

FIGURE 2.7 – Découpage de l'intervalle de codage dans le cas le plus général du CA avec SI multiples.

L'algorithme de codage arithmétique décrit dans le paragraphe 2.2.2.3 est le même pour les CAs avec SI et SI multiple, mais avec une source dont l'alphabet est étendu $\mathcal{A}^e = \mathcal{A} \cup \{\varepsilon_1, \varepsilon_2, \cdots \varepsilon_M, \varepsilon_{M+1}\}$. Par contre seuls les éléments de $\mathcal{A}$ sont générés par la source X.

Pour le codage arithmétique de base (en *précision infinie*), il est très difficile pour un calculateur à *précision finie* de représenter exactement tous les nombres réels (bornes et largeurs des sous-intervalles par exemple). Ce problème est traité avec le CA en précision finie [Pas76, Ris76] ou *code quasi-arithmétique* (CQA) [HV92].

2.2.2.5 Codage arithmétique en précision finie

Le CA en précision finie [Pas76, HV92, Ris76] s'effectue de la même manière que l'implémentation pratique du codage arithmétique de base décrite dans le paragraphe 2.2.2.3, mais l'intervalle initial de codage $[0, 1)$ est remplacé par un intervalle entier $[0, T)$ où $T = 2^q$ et q est la précision binaire du dispositif de codage. Par conséquent FIRST_QTR $= \frac{T}{4}$, HALF $= \frac{T}{2}$ et THIRD_QTR $= \frac{3T}{4}$ (voir tableau 2.1, page 47 pour les détails de codage). Ici, lors du processus de codage, les valeurs l et h sont arrondies aux entiers les plus proches. Arrondir les valeurs de l et h a pour effets d'introduire plus de redondance et ces redondances sont d'autant plus importantes que la précision q utilisée est faible. L'introduction des SI et SI multiples décrite au paragraphe 2.2.2.4 s'applique également au CA en précision finie.

2.2.3 Propriétés de distance des codes conjoints

Dans ce paragraphe, nous allons rappeler la définition des *propriétés de distance* (la d_{libre} et le spectre de distance) des CLVs conjoints et CAs conjoints. Pour définir ces paramètres, une *représentation graphique* de ces codeurs est plus adaptée. En effet, les CLVs et CAs conjoints peuvent être générés par des *"codeurs à états finis"* (CEFs) à *longueur variable* (CEF-LVs) qui génèrent des *codes à états finis* à longueur variable voir [WK10].

2.2.3.1 Codeur à états finis à longueur variable

Un codeur à états finis à LV peut décrit par un graphe orienté $\Gamma(\mathcal{S}, \mathcal{T})$, où $\mathcal{S}$ désigne l'ensemble des états (appelé aussi sommets ou noeuds) et $\mathcal{T}$ l'ensemble des transitions (arcs orientés). Chaque transition $u \in \mathcal{T}$ est étiquetée avec une séquence de symboles d'entrée $I(u)$ et une séquence de bits de sortie $O(u)$ qui peuvent être de longueurs variables. Soient $\sigma(u)$ et $\tau(u)$ les états de départ et d'arrivée de la transition u. $P(u)$, $u \in \mathcal{T}$ est la probabilité que la séquence $I(u)$ soit émise par la source.

Un chemin $\boldsymbol{u} = (u_1 \circ u_2 \circ \cdots \circ u_k) \in \mathcal{T}^k$ dans le graphe Γ est une concaténation de transitions qui satisfont $\sigma(u_{i+1}) = \tau(u_i)$, pour $1 \leqslant i \leqslant k-1$ (ceci correspond à un parcours de longueur k dans le graphe du codeur). Par extension, on définit $\sigma(\boldsymbol{u}) = \sigma(u_1)$ et $\tau(\boldsymbol{u}) = \tau(u_k)$. De plus $I(\boldsymbol{u})$ et $O(\boldsymbol{u})$ sont les concaténations des

étiquettes d'entrée et de sortie respectives de $\boldsymbol{u}$. La probabilité du chemin pour une source sans mémoire est $P(\boldsymbol{u}) = \prod_{i=1}^{k} P(u_i)$. Finalement, $\ell(x)$ est la longueur (en symboles ou en bits) de la séquence $\boldsymbol{x}$.

On considère que le graphe du codeur à états finis à longueur variable est *irréductible*, c'est-à-dire que tout état peut être atteint à partir d'un autre état avec un nombre fini de transitions. On suppose aussi que le graphe est *apériodique*, c'est-à-dire que les temps de récurrence des états mesurés en nombre de bits de sortie ne sont pas multiples d'un nombre entier $m > 1$. Ces hypothèses impliquent que le codeur à états finis à longueur variable et la source à coder, forment une chaine de Markov avec une distribution stationnaire d'état unique. Soit $P^*(s)$, $s \in \mathcal{S}$ la probabilité stationnaire de l'état s, c'est-à-dire la probabilité pour la machine à états finis d'être dans l'état s en régime permanent. Cette probabilité est calculée en prenant en compte les longueurs des étiquettes de sortie comme présenté dans [WK10].

2.2.3.2 Application au code à longueur variable conjoint

Dans le cas le plus simple d'un CLV conjoint associé à un alphabet $\mathcal{A}_M$ et un dictionnaire $\mathcal{C}_M$, $\mathcal{S}$ ne contient qu'un seul état s_0 qui est l'état de départ et d'arrivé de toutes les transitions, et M transitions associées aux éléments de $\mathcal{A}_M$. La transition $u_i : 1 \leqslant i \leqslant M$ a pour étiquette d'entrée $I(u_i) = a_i$, pour étiquette de sortie $O(u_i) = c_i$ et une probabilité associée $P(u_i) = p_i$.

Exemple 2.6 *La figure 2.8 montre un exemple d'CEF-LV associé à la source X_3 à valeurs dans d'alphabet $\mathcal{A}_3 = \{a, b, c\}$ codée en avec le dictionnaire $\mathcal{C}_3 = \{0, 10, 111\}$.*

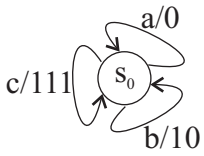


FIGURE 2.8 – Exemple d'un codeur à états finis à longueur variable associé à $\mathcal{A}_3 = \{a, b, c\}$ et $\mathcal{C}_3 = \{0, 10, 111\}$

2.2.3.3 Application au code arithmétique conjoint

Le CA en précision finie quant à lui peut être défini par les *paramètres du codeur* $\{T, f_{\max}, \boldsymbol{p}, p_\epsilon\}$. A partir de ces paramètres, le processus de codage peut être caractérisé par $[l, h)$ (qui appartient à un ensemble fini), f et les probabilités de la source. Par conséquent, l'état de l'automate qui représente le codeur peut être caractérisé comme (l, h, f). Si la valeur de f est bornée, il est possible de générer tous les états possibles et les transitions entre les états, ce qui conduit à un codeur à états finis. En général, la valeur de f peut croître à l'infini, mais peut facilement être limitée à $f \leqslant f_{\max}$, comme dans [BHS06], d'où le code quasi arithmétique (CQA). Cette thèse considère l'approche de [BJWK08] : à chaque fois que $f = f_{\max}$ et que l'intervalle courant de codage est tel que f peut encore être incrémentée, les probabilités des symboles sont temporairement modifiées pour forcer une sortie de bit après le codage du symbole courant.

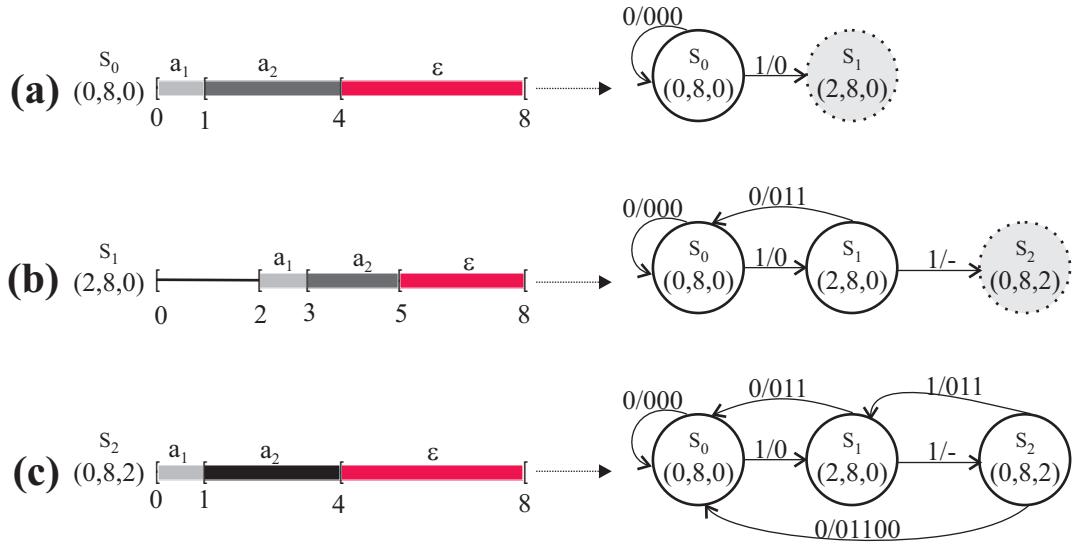


FIGURE 2.9 – Exemple de génération d'un codeur à états finis à longueur variable associé à une source à deux symboles codée avec un CA en précision finie dont les paramètres sont $T = 8$, $f_{\max} = 2$, $p_0 = 1/4$, $p_\epsilon = 1/2$

Exemple 2.7 *Considérons l'alphabet $\mathcal{A}_2 = \{a_1 = 0, a_2 = 1\}$ à encoder en utilisant le CA en précision finie dont les paramètres sont $T = 8$, $f_{\max} = 2$, $p_0 = 1/4$, $p_\epsilon = 1/2$. La figure 2.9 montre les étapes de la construction d'un codeur à états finis à LV associé à la source X_2 à partir de l'état initial $s_0 = (l = 0, h = 8, f = 0)$.*

La première étape correspondante à la figure 2.9 (a) montre une subdivision possible de l'intervalle associé à l'état initial dans laquelle on assigne au symbole '0' *le sous-intervalle* $[0, 1)$ *qui conduit au même état initial avec la transition* $u_1 = 0/000$ *; et on assigne au symbole* '1' *le sous-intervalle* $[1, 4)$ *qui conduit à un nouvel état* $s_1 = (2, 8, 0)$ *avec la transition* $u_2 = 1/0$*. La figure 2.9 (b) montre une subdivision possible de l'intervalle correspondant à l'état* $s_1 = (2, 8, 0)$*. Le sous-intervalle* $[2, 3)$ *est associé au symbole* '0' *et le sous-intervalle* $[3, 5)$ *est associé au symbole* '1' *conduisant respectivement à l'état* s_0 *avec la transition* $u_3 = 0/011$ *et à un nouvel état* $s_2 = (0, 8, 1)$ *avec la transition* $u_4 = 1/-$*. La subdivision de l'intervalle correspondant à ce nouvel état est présentée à la figure 2.9 (c) qui ne conduit qu'à des états déjà connus avec les transitions* $u_5 = 0/01100$ *et* $u_6 = 1/011$ *; donc on obtient l'automate complet qui décrit le codeur quasi-arithmétique.*

2.2.4 Propriétés des codes conjoints

Dans [BJWK08], trois types de codeur à états finis décrivant le processus de codage ont été proposés.

1. Le *codeur à états finis synchronisé symbole* adapté pour le codage, où l'étiquette d'entrée de chaque transition correspond exactement à un seul symbole.
2. Le *codeur à états finis réduit* avec des transitions à étiquettes d'entrée et de sortie à longueurs variables et non vides, conduisant à un treillis compact adapté pour le décodage et l'évaluation des performances de compression. Il est obtenu à partir du codeur à états finis synchronisé symbole dans le quel toutes les transitions qui ont des étiquettes de sortie vides sont étendues jusqu'à avoir au moins un bit. Cela implique que des transitions dans le codeur à états finis réduit auront des étiquettes d'entrées composées de plus d'un symbole. Dorénavant nous désignons tout simplement par codeur à états finis réduit par codeur à états finis. Dans la suite $\mathcal{S}$ et $\mathcal{T}$ représentent l'ensemble des états et des transitions dans le codeur à états finis.
3. Le *codeur à états finis synchronisé bit* adapté pour l'évaluation du spectre de distance, où l'étiquette de sortie de chaque transition contient un seul bit. Il est obtenu à partir du codeur à états finis dans lequel, une transition u_i dont l'étiquette de sortie contient n bits est divisée en n transitions u_i^j pour

$1 \leqslant j \leqslant n$. La transition u_i^1 hérite de $I(u_i)$ comme étiquette d'entrée et du premier bit de $O(u_i)$ comme étiquette de sortie, et chaque transition u_i^j pour $2 \leqslant j \leqslant n$ a une étiquette d'entrée vide et a pour étiquette de sortie le $j^{\text{ième}}$ bit de $O(u_i)$. Cela a pour conséquence d'introduire $n-1$ états symboliques supplémentaires. Pour la suite, $\mathcal{S}_b$ et $\mathcal{T}_b$ désignent l'ensemble des états et de transitions dans le codeur à états finis synchronisé bit.

La figure 2.10 (a) représente le codeur à états finis synchronisé symbole et le codeur à états finis du code à longueur variable conjoint de l'exemple 2.6 (à la page 51) et la figure 2.10 (b) représente le codeur à états finis synchronisé bit associé.

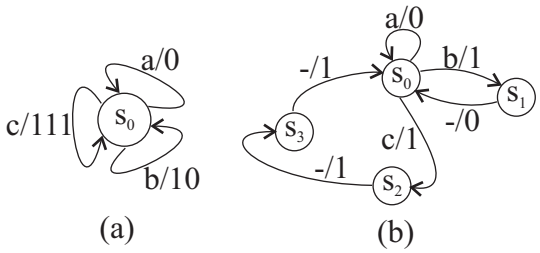


FIGURE 2.10 – (a) Codeur à états finis et Codeur à états finis synchronisé symbole de l'exemple 2.6 et (b) le Codeur à états finis synchronisé bit associé.

La figure 2.11 (a) correspond au codeur à états finis synchronisé symbole de l'exemple 2.7 et la figure 2.11 (b) représente le codeur à états finis associé.

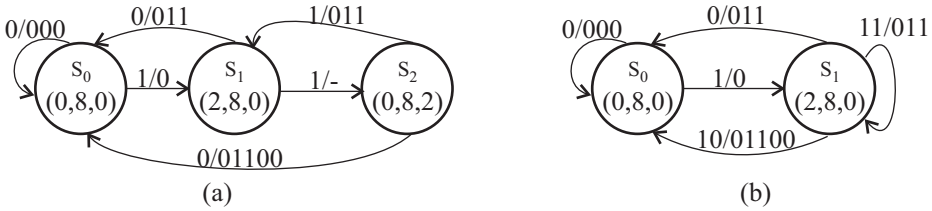


FIGURE 2.11 – (a) Codeur à états finis synchronisé symbole de l'exemple 2.7 et (b) le codeur à états finis associé.

A partir de l'état initial $s_0 \in \mathcal{S}$, la succession des états de le codeur à états finis à longueur variable pour toutes les séquences d'entrée possibles de longueur (semi-)infinie peut être représentée dans un treillis qui décrit l'évolution temporelle de l'état du codeur à états finis à LV. Les étiquettes de sortie de tous les chemins

à travers le treillis forment le code à états finis à LV dont les performances sont caractérisées par son *taux de codage* R_c et par sa *capacité de correction d'erreurs*. Comme pour les codes de canal linéaires, la performance de correction d'erreurs est caractérisé d'abord par la *distance libre* d_{libre}, une caractérisation plus fine est possible à travers le *spectre de distance* [VO79]

Definition 2.12 *Le taux de codage R_c en symboles par bit est le rapport entre la longueur moyenne des étiquettes d'entrée et la longueur moyenne des étiquettes de sortie des transitions dans $\mathcal{T}$,*

$$R_c = \frac{\sum_{u\in\mathcal{T}} P^*(\sigma(u))P(u)\ell(I(u))}{\sum_{u\in\mathcal{T}} P^*(\sigma(u))P(u)\ell(O(u))}, \tag{2.29}$$

Definition 2.13 *Soit $\mathcal{U}_{s_0}^k$ l'ensemble des chemins avec k transitions qui partent de l'état s_0. Le code à états finis à LV $\mathcal{C}(\Gamma, s_0)$, est l'ensemble de toutes les séquences de sortie de longueur infinie générées par le codeur à états finis à LV à partir de l'état initial s_0, $\mathcal{C}(\Gamma, s_0) = \{O(\boldsymbol{u}) : \boldsymbol{u} \in \mathcal{U}_{s_0}^\infty\}$.*

Considérons deux chemins $(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \mathcal{T}^{k_1} \times \mathcal{T}^{k_2}$ tel que $\ell(O(\boldsymbol{u}_1)) = \ell(O(\boldsymbol{u}_2))$, alors on notera $d_H(\boldsymbol{u}_1, \boldsymbol{u}_2) = d_H(O(\boldsymbol{u}_1), O(\boldsymbol{u}_2))$.

Definition 2.14 *La distance libre, d_{libre}, du code $\mathcal{C}(\Gamma, s_0)$ est la distance minimale de Hamming entre toutes les paires de séquences sur différents chemins. Soit $\mathcal{U}$ l'ensemble des paires de chemins dans $\left(\mathcal{T}^{k_1} \times \mathcal{T}^{k_2}\right)_{1\le k_1,k_2<\infty}$ qui divergent d'un état et qui convergent pour la première fois dans le même ou dans un autre état avec le même nombre de bits sur leur étiquettes de sortie. Alors, d_{libre} est aussi la distance minimale de Hamming dans $\mathcal{U}$*

$$d_{\text{libre}} = \min_{\substack{(\boldsymbol{u}_1,\boldsymbol{u}_2)\in\mathcal{U} \\ \boldsymbol{u}_1\neq\boldsymbol{u}_2}} d_H\left(\boldsymbol{u}_1, \boldsymbol{u}_2\right). \tag{2.30}$$

Definition 2.15 *Le spectre de distance [Vit71] dans le domaine code peut être représenté par la fonction génératrice,*

$$G\left(D\right) = \sum_{d=d_{\text{libre}}}^{\infty} A_d D^d, \tag{2.31}$$

où A_d est le nombre moyen de chemins à distance de Hamming d d'un chemin donné. Dans le cas le plus général, A_d peut être défini comme [BJWK08, WK10],

$$A_d = \sum_{\substack{(\boldsymbol{u}_1,\boldsymbol{u}_2)\in\mathcal{U} \\ d_H(\boldsymbol{u}_1,\boldsymbol{u}_2)=d}} P^*(\sigma(\boldsymbol{u}_1))P(\boldsymbol{u}_1) \tag{2.32}$$

Avec les paramètres définis ci-dessus, on constate que le code $\mathcal{C}(\Gamma, s_0)$ et sa distance libre sont indépendants de la source (à condition que tous les symbole de la sources ont une probabilité non nulle), alors que le caractère source-canal conjoint se montre dans le fait que le taux de codage et les coefficients du spectre dépendent des statistiques de la source.

2.3 Conclusion

Dans ce chapitre, nous avons rappelé les propriétés de distance des codes de canal linéaires et nous avons défini les propriétés de distance des codes conjoints non-linéaires. Le chapitre 3 sera consacré à l'évaluation des propriétés de distance de ces codes conjoints non-linéaires.

Chapitre 3

Évaluation des propriétés de distance de codes non-linéaires

Ce chapitre est essentiellement consacré à l'évaluation des propriétés de distance des codes conjoints non-linéaires décrits par des codeurs à états fins à longueur variable (CEF-LV). La méthode développée dans cette thèse est décrite dans le paragraphe 3.1. Dans le paragraphe 3.2, nous comparons la complexité de cette méthode avec celles de quelques méthodes disponibles dans la littérature.

3.1 Évaluation des propriétés de distance

A partir d'ici, seul le codeur à états finis synchronisé bit sera considéré. Pour évaluer la distance libre et le spectre de distance d'un code à états finis décrit par un codeur à états finis synchronisé bit $\Gamma_b(\mathcal{S}_b, \mathcal{T}_b)$, des techniques inspirées de [Big84] sont appliquées pour suivre les distances de Hamming entre les paires de chemins dans le codeur à états finis synchronisé bit.

Pour cela, on considère le graphe produit $\Gamma_b^2 = \Gamma_b \times \Gamma_b$ étiqueté avec les distances de Hamming des paires de transitions dans $\mathcal{T}_b^2$ (ce qui conduit au treillis produit considéré dans [Big84]). La différence principale avec [Big84] est que pour le CEF-LV, les états peuvent avoir un nombre de transitions variable. Ensuite, on montre comment ce graphe produit peut être simplifié.

3.1.1 Génération du graphe produit

Ici, nous décrivons la génération du graphe produit associé au codeur à états finis synchronisé bit et sa simplification pour calculer efficacement la distance libre du code à états finis associé.

Considérons l'ensemble des états $\mathcal{S}_b = \{s_i \ : \ 0 \leqslant i < |\mathcal{S}_b|\}$ (où $|\mathcal{A}|$ est le cardinal de l'ensemble $\mathcal{A}$) et l'ensemble des transitions $\mathcal{T}_b$ du graphe orienté $\Gamma_b(\mathcal{S}_b, \mathcal{T}_b)$ représentant un codeur à états finis synchronisé bit. Le graphe produit associé à $\Gamma_b(\mathcal{S}_b, \mathcal{T}_b)$ est le graphe orienté $\Gamma_b^2(\mathcal{S}_b \times \mathcal{S}_b, \mathcal{T}_b \times \mathcal{T}_b)$ avec $|\mathcal{S}_b|^2$ états $s_{i,j}$ définis comme

$$s_{i,j} = (s_i, s_j),\ 0 \leqslant i, j < |\mathcal{S}_b|. \tag{3.1}$$

Pour toute paire de transitions (u, v) dans le graphe original, Γ_b^2 contient un arc orienté

$$e = (u, v) : \sigma(e) = s_{\sigma(u),\sigma(v)} \text{ et } \tau(e) = s_{\tau(u),\tau(v)}. \tag{3.2}$$

Le poids de Hamming de l'arc e, $w_{\mathrm{H}}(e)$ est la distance de Hamming entre la sortie des deux transitions u et v, c'est-à-dire :

$$w_{\mathrm{H}}(e) = d_{\mathrm{H}}\left(O(u), O(v)\right). \tag{3.3}$$

Un chemin orienté $\boldsymbol{e}$ dans Γ_b^2, qui part de l'état $\mathrm{s}_{i,j}$ et arrive à l'état $\mathrm{s}_{m,n}$, est une succession d'arcs $\boldsymbol{e} = (e_1 \circ e_2 \circ \cdots \circ e_N)$ tel que $\sigma(e_{\mu+1}) = \tau(e_\mu)$ pour $1 \leqslant \mu < N$. Le poids de ce chemin orienté, $w_{\mathrm{H}}(\boldsymbol{e})$ est défini comme :

$$w_{\mathrm{H}}(\boldsymbol{e}) = \sum_{\mu=1}^{N} w_{\mathrm{H}}(e_\mu). \tag{3.4}$$

Comme Γ_b est synchronisé bit, c'est-à-dire que l'étiquette de sortie de chaque transition contient exactement un seul bit, on a $w_{\mathrm{H}}(\boldsymbol{e}) \leqslant N$. Ainsi, le poids du chemin orienté dans Γ_b^2 de l'état $\mathrm{s}_{i,j}$ à un état $\mathrm{s}_{m,n}$, est la distance de Hamming entre les bits de sortie des deux chemins : $(\boldsymbol{u}_1, \boldsymbol{u}_2) \in \mathcal{T}_b^k \times \mathcal{T}_b^k$.

Par conséquent, lorsqu'on parcourt le graphe produit à partir de l'état initial $s_{0,0}$, les poids des chemins orientés obtenus représentent toutes les distances de Hamming de toutes les paires de séquences possibles dans $\mathcal{C}\left(\Gamma_b, s_0\right) \times \mathcal{C}\left(\Gamma_b, s_0\right)$, y compris la distance libre selon la définition 2.14 (voir paragraphe 2.2.3 à la page 55).

La figure 3.1 montre l'exemple d'un codeur à états finis synchronisé bits ; s_0 et s_2 sont les états d'où des paires de chemins peuvent diverger.

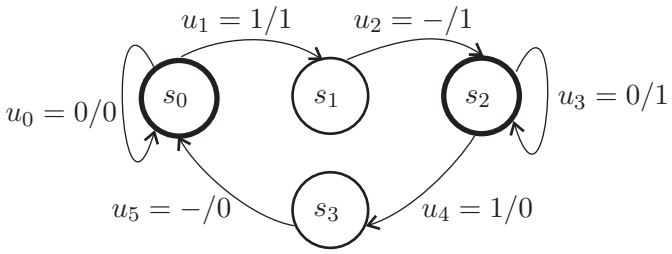


FIGURE 3.1 – Exemple d'un codeur à états finis synchronisé bits

La figure 3.2 montre le graphe produit associé au codeur de la figure 3.1. Les arcs dans la figure 3.2 sont étiquetés avec leur poids de Hamming. Par exemple, sur le graphe produit l'arc e_0 qui pat de l'état $s_{0,0}$ et qui arrive dans le même état $s_{0,0}$ correspond à la paire de transitions (u_0, u_0) dans $\mathcal{T}_b^2$, donc on a :

$$w_{\mathrm{H}}(e_0) = d_{\mathrm{H}}(u_0, u_0) = 0, \tag{3.5}$$

tandis que l'arc e_1 qui part de l'état $s_{0,0}$ et qui arrive dans l'état $s_{0,1}$ correspond à la paire (u_0, u_1) qui conduit à :

$$w_{\mathrm{H}}(e_1) = d_{\mathrm{H}}(u_0, u_1) = d_{\mathrm{H}}(0, 1) = 1. \tag{3.6}$$

Dans le graphe produit, le chemin $\boldsymbol{e}_0$ en pointillés gras défini par la succession d'états $(s_{0,0}, s_{1,0}, s_{2,1}, s_{2,2})$, correspond dans le graphe original de la figure 3.1 à la paire de chemins $(\boldsymbol{u}_1, \boldsymbol{u}_2)$ définis comme : $\boldsymbol{u}_1 = (u_1 \circ u_2 \circ u_3)$ et $\boldsymbol{u}_2 = (u_0 \circ u_1 \circ u_2)$. Par conséquent

$$w_{\mathrm{H}}(\boldsymbol{e}_0) = d_{\mathrm{H}}(\boldsymbol{u}_1, \boldsymbol{u}_2) = d_{\mathrm{H}}(111, 011) = 1. \tag{3.7}$$

3.1.2 graphe produit modifié

D'après la définition 2.14 (voir la page 55), l'évaluation de la distance libre (d_{libre}) n'implique que les chemins dans Γ_b^2 qui appartiennent à $\mathcal{U}$ (c'est-à-dire l'ensemble des paires de chemins dans $\left(\mathcal{T}_b^k \times \mathcal{T}_b^k\right)_{1 \leq k < \infty}$ qui divergent d'un état et qui convergent pour la première fois dans le même ou dans un autre état avec le même nombre de bits sur leur étiquettes de sortie). Donc on peut déduire de Γ_b^2 un *graphe produit modifié* qui ne représente que ces chemins.

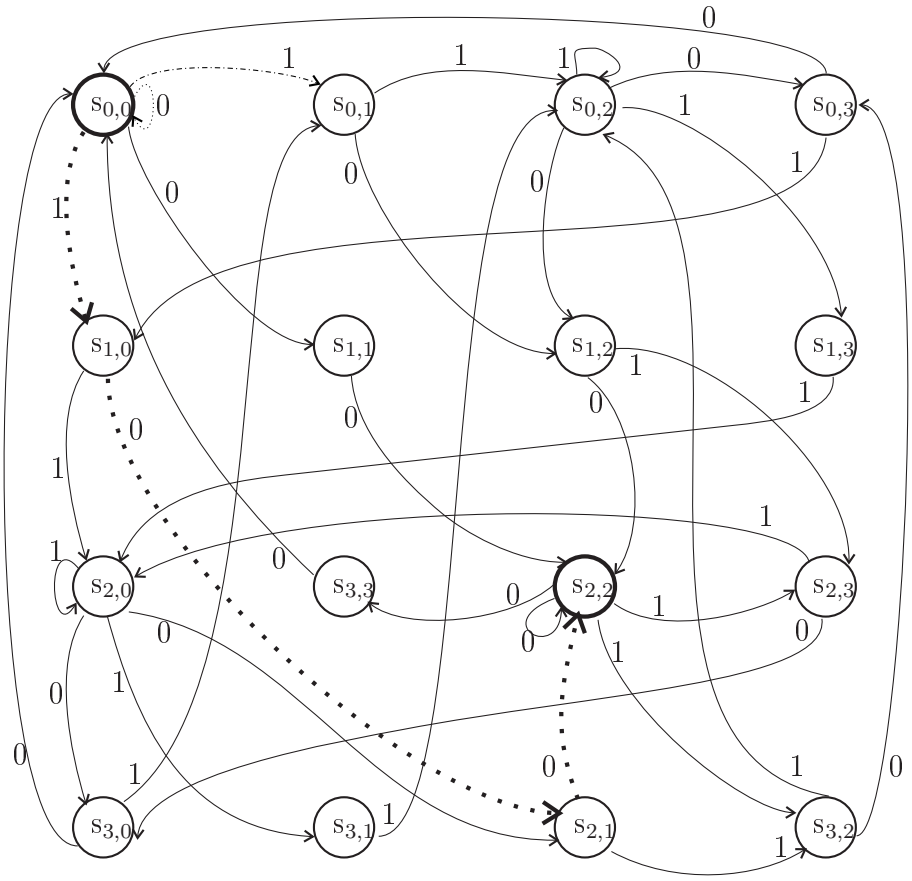


FIGURE 3.2 – Graphe produit dérivé du codeur de la figure 3.1

Soient les deux ensembles d'états $S_{\text{div}} \subset \mathcal{S}_b^2$ et $S_{\text{conv}} \subset \mathcal{S}_b^2$ tels que

$$\mathcal{S}_{\text{div}} = \left\{ s_{i,i} \in \mathcal{S}_b^2 \ : \ \exists\, (u,v) \in \mathcal{T}_b^2, u \neq v, \text{tel que } \sigma(u) = \sigma\,(v) = s_i \right\}, \tag{3.8}$$

$$\mathcal{S}_{\text{conv}} = \left\{ s_{i,i} \in \mathcal{S}_b^2 \ : \ \exists\, (u,v) \in \mathcal{T}_b^2, u \neq v, \text{tel que } \tau(u) = \tau(v) = s_i \right\}. \tag{3.9}$$

$\mathcal{S}_{\text{div}}$ est l'ensemble des états de Γ_b^2 dont les arcs sortants sont composés de paires de transitions distinctes dans $\mathcal{T}_b^2$ et qui divergent d'un même état dans $\mathcal{S}_b$. On regroupe les états de $\mathcal{S}_{\text{div}}$ dans un seul état s_{in} avec seulement des arcs sortants. $\mathcal{S}_{\text{conv}}$ est l'ensemble des états dans Γ_b^2 dans lequel les arcs entrants sont composés de paires de transitions distinctes dans $\mathcal{T}_b^2$ et qui convergent dans le même état d'arrivée dans $\mathcal{S}_b$. On regroupe les états de $\mathcal{S}_{\text{conv}}$ dans un seul état s_{out} avec seulement des arcs entrants. Dans la figure 3.2, $\mathcal{S}_{\text{div}} = \{s_{0,0}, s_{2,2}\}$ et $\mathcal{S}_{\text{conv}} = \{s_{0,0}, s_{2,2}\}$.

L'ensemble des arcs $\{e = (u,u) \ : \ u \in \mathcal{T}_b\}$ dans Γ_b^2 correspond à des paires de chemins qui n'ont jamais divergé, par conséquent selon la définition 2.14, cet en-

semble ne sera pas utile pour trouver d_{libre}. Si on remplace dans Γ_b^2 les états $\mathcal{S}_{\text{div}}$ et $\mathcal{S}_{\text{conv}}$ par s_{in} et s_{out} respectivement, et on enlève l'ensemble $\{e = (u, u) \ : \ u \in \mathcal{T}_b\}$, on obtient un *graphe produit modifié*, dans lequel s_{in} est l'état initial et s_{out} est l'état final, et qui contient tous les chemins nécessaires à l'évaluation de d_{libre}. Le graphe produit modifié obtenu à partir du graphe produit de la figure 3.2 est présenté dans la figure 3.3. Lorsqu'on parcourt le graphe produit modifié de l'état initial à l'état

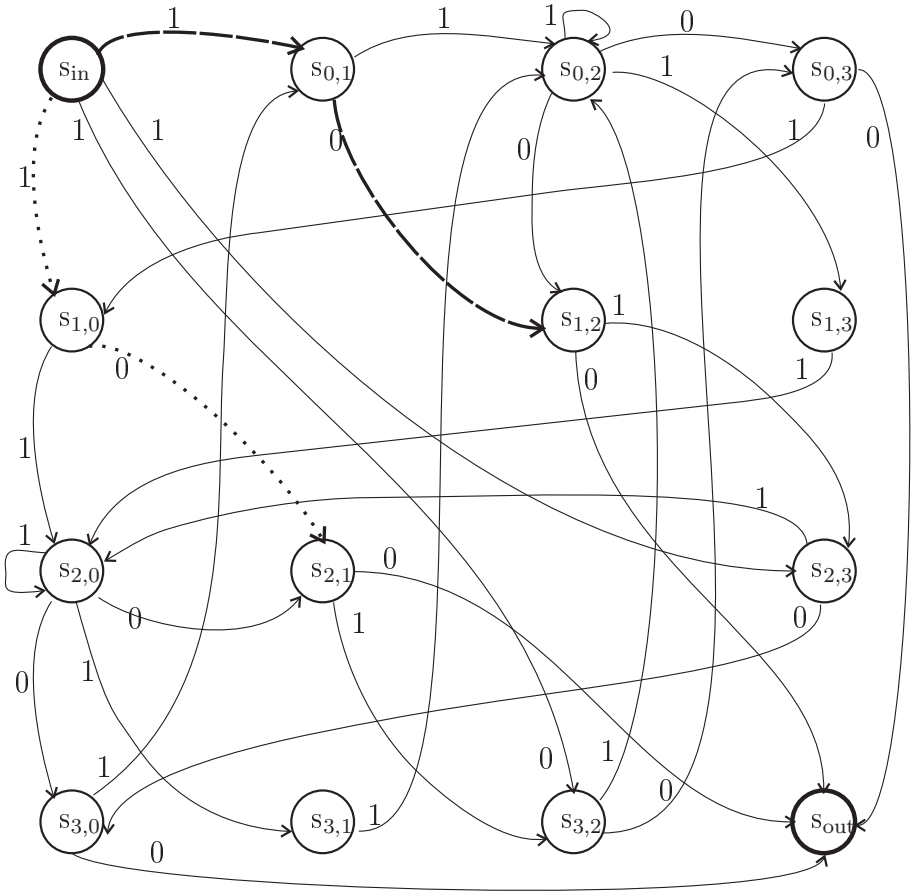


FIGURE 3.3 – Graphe produit modifié déduit du graphe produit de la figure 3.2

final, les poids des chemins donnent les distances de Hamming dans $\mathcal{U}$. Ainsi, trouver d_{libre} revient à déterminer un chemin orienté entre l'état initial à l'état final avec le plus petit poids de Hamming.

3.1.3 Graphe des distances entre paires

Par ailleurs, dans le graphe produit modifié, si $\boldsymbol{e}_1$ est un chemin direct entre s_{in} et $s_{i,j}$, $i \neq j$, il existe aussi un chemin direct $\boldsymbol{e}_2$ entre s_{in} à $s_{j,i}$ tel que

$$\ell(\boldsymbol{e}_1) = \ell(\boldsymbol{e}_2) \;\text{ et }\; w_{\text{H}}(\boldsymbol{e}_1) = w_{\text{H}}(\boldsymbol{e}_2). \tag{3.10}$$

Ceci est du au fait que d_{H} est symétrique par rapport à ses arguments. Les chemins $\boldsymbol{e}_1$ et $\boldsymbol{e}_2$ dans le graphe produit modifié sont équivalents en terme de poids de Hamming. Par exemple sur le graphe produit modifié de la figure 3.3, le chemin $\boldsymbol{e}_1$ en *tirets* entre l'état s_{in} et l'état $s_{1,2}$; et le chemin $\boldsymbol{e}_2$ en *pointillés* entre les états s_{in} et $s_{2,1}$ sont tous les deux de longueur 2 et de poids $w_{\text{H}}(\boldsymbol{e}_1) = w_{\text{H}}(\boldsymbol{e}_2) = 1$ et sont équivalents.

Ainsi, la complexité de recherche du plus court chemin peut être réduite en définissant le *graphe des distances entre paires* (GDP) qui contient un seul chemin pour toute paire de chemins équivalents dans le graphe produit modifié. Pour construire ce graphe, les deux états $s_{i,j}$ et $s_{j,i}$ dans le graphe produit modifié sont fusionnés et remplacés par un seul état $s_{\nu,\gamma}$ ($\nu = \min(i,j), \gamma = \max(i,j)$) dans le graphe des distances entre paires, et les deux arcs orientés (u,v) et (v,u) (avec $u \in \mathcal{T}_b$ et $v \in \mathcal{T}_b$) dans le graphe produit modifié sont remplacés par un seul arc dans le graphe des distances entre paires.

Par conséquent, trouver d_{libre} avec ce GDP revient à trouver un chemin orienté entre s_{in} et s_{out} avec le plus petit poids de Hamming. En théorie des graphes, ceci est connu sous le nom de *problème du plus court chemin* et peut être résolu efficacement avec l'algorithme de Dijkstra [GM84], puisque tous les poids sont non négatifs.

La figure 3.4 représente le GDP associé au graphe produit modifié de la figure 3.3.

Le nombre d'états $|\mathcal{S}_{\text{GDP}}|$ et le nombre d'arcs $|\mathcal{T}_{\text{GDP}}|$ dans le graphe des distances entre paires (GDP), sont

$$|\mathcal{S}_{\text{GDP}}| = \frac{|\mathcal{S}_b| \times (|\mathcal{S}_b| - 1)}{2} + 2, \tag{3.11}$$

$$|\mathcal{T}_{\text{GDP}}| = \frac{|\mathcal{T}_b| \times (|\mathcal{T}_b| - 1)}{2}, \tag{3.12}$$

$|\mathcal{S}_{\text{GDP}}|$ et $|\mathcal{S}_{\text{GDP}}|$ sont les nombres maximum d'états et d'arcs atteints lorsque l'algorithme de Dijkstra est appliqué sur le graphe des distances entre paire pour calculer la distance libre.

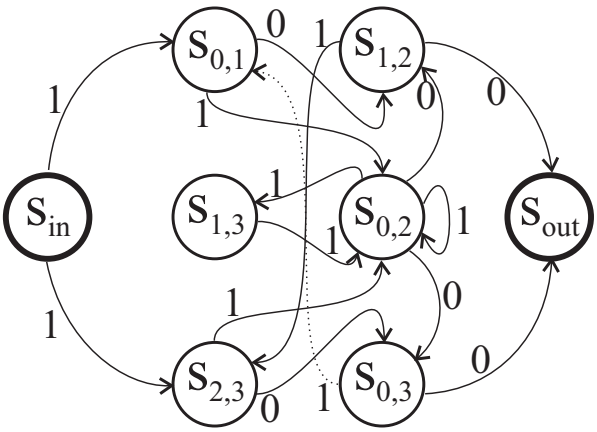


FIGURE 3.4 – Le graphe des distances entre paires de chemins

3.1.4 Spectre des distances

Ici, nous allons décrire brièvement comment le graphe produit modifié pourrait être étiqueté de telle sorte que le spectre de distance dans le domaine code soit obtenu à partir d'une fonction génératrice. Cette dernière pourrait être évaluée en utilisant une fonction de transfert symbolique, par exemple, en utilisant la formule du gain de Mason [Mas56].

Pour cela, on associe à chaque arc $e = (u, v)$ un gain $g(e)$ défini comme

$$g(e) = \begin{cases} P^*(\sigma(u))P(u)D^{w_{\mathrm{H}}(e)} \text{ si } \sigma(u) = \sigma(v), \\ P(u)D^{w_{\mathrm{H}}(e)} \quad \text{si } \sigma(u) \neq \sigma(v), \end{cases} \tag{3.13}$$

où D est une variable symbolique utilisé pour indiquer le poids de Hamming du chemin. Ensuite, on applique la formule du gain de Mason entre s_{in} et s_{out} dans le graphe produit modifié permettant d'obtenir la fonction de transfert $G(D)$, qui représente le spectre de distance défini dans (2.32).

3.2 Comparaison de complexité

Comme nous l'avons déjà précisé, une méthode alternative pour calculer la distance libre est présentée dans [BJWK08]. Cette méthode a été développée pour les codes arithmétiques conjoints, mais elle peut s'appliquer à tous les codes à états finis à longueur variable ; les codes à états finis à longueur fixes sont des cas particuliers

des codes à états finis à longueur variable.

Pour ce calcul de la distance libre, [BJWK08] utilise une matrice tri-dimensionnelle définie comme $\Delta_n = (d_{k,i,j})$, où $d_{k,i,j}$ est la distance minimale de Hamming entre toutes les paires de chemins de longueur n dans le codeur à états finis synchronisé bits qui partent de l'état s_k et finissent respectivement dans les états s_i et s_j sans avoir déjà convergé ; et n est la longueur des chemins. Cette méthode est itérative sur la longueur n du chemin. Elle compare les chemins pour des longueurs allant jusqu'à n_{libre}, qui est la longueur à partir de laquelle toutes les paires de chemins qui n'ont pas encore convergé dans un même état sont à une distance de Hamming supérieure ou égale à la distance libre.

En pratique, l'implémentation de l'algorithme de [BJWK08] nécessite de fixer une borne supérieure $n_{\max}$ sur n pour l'exécuter dans un temps fini. Si $n_{\max}$ est trop petit, il n'est pas garanti de trouver la vraie valeur de d_{libre}. Ce qui peut arriver avec les codes catastrophiques, puisque ces codes contiennent des paires de chemins à une distances de Hamming finie et qui ne convergent jamais entre eux. La complexité de cette méthode est de $\mathcal{O}(n_{\max} \times |\mathcal{T}_b|^2 \times |\mathcal{S}_r|)$.

La pire complexité en appliquant l'algorithme de Dijkstra sur le graphe des distances entre paires (GDP) est $|\mathcal{S}_{\text{GDP}}|^2$. Avec une implémentation meilleure utilisant les tas de Fibonacci, la complexité peut être réduite à $O(|\mathcal{T}_{\text{GDP}}| + |\mathcal{S}_{\text{GDP}}| \times \log(|\mathcal{S}_{\text{GDP}}|))$.

L'existence d'une paire de chemins catastrophique implique un circuit de poids nul dans le graphe produit et vice-versa. Par exemple, dans la figure 3.5 (a) les séquences obtenues par $(s_0, s_1, s_3, \cdots, s_3)$ et $(s_0, s_2, s_4, \cdots, s_4)$ sont à une distance de Hamming de un. L'algorithme de Dijkstra n'a pas de problème de terminaison avec ces codes car pendant le processus, chaque arc n'est exploré qu'une seule fois au maximum.

Une autre méthode qui calcule directement le spectre de distance (dans le domaine code) des codes à états finis à longueur variable en utilisant des matrices à la place d'une fonction de transfert sur un graphe de fluence est proposée dans [WK10]. Dans cette méthode, les coefficients A_d de la série formelle $G(D)$, (pour $0 \leqslant d < \infty$ voir (2.31) et (2.32) à la page 55) sont calculés un à un. L'indice du premier coefficient non nul donne la distance libre. La méthode dans [WK10] utilise une inversion de matrice avec des matrices de taille $|\mathcal{S}_b|^2 \times |\mathcal{S}_b|^2$ pour calculer les

coefficients. En utilisant par exemple l'élimination de Gauss, cette méthode à un complexité de $|\mathcal{S}_b|^{4\times 3}$.

Cette méthode est très efficace puisqu'elle permet d'obtenir les vraies valeurs des coefficients, donc la vraie valeur de d_{libre}, mais elle reste très complexe pour effectuer une recherche de code. Car pour cette recherche, on a en premier lieu besoin de calculer assez rapidement d_{libre} ; on a besoin ni de $A_{d_{\text{libre}}}$ ni du reste du spectre. Or, la méthode définie dans [WK10] risque de calculer un certain nombres de termes A_d $(1 \leqslant d < d_{\text{libre}})$ avant de trouver $A_{d_{\text{libre}}}$, donc elle reste complexe.

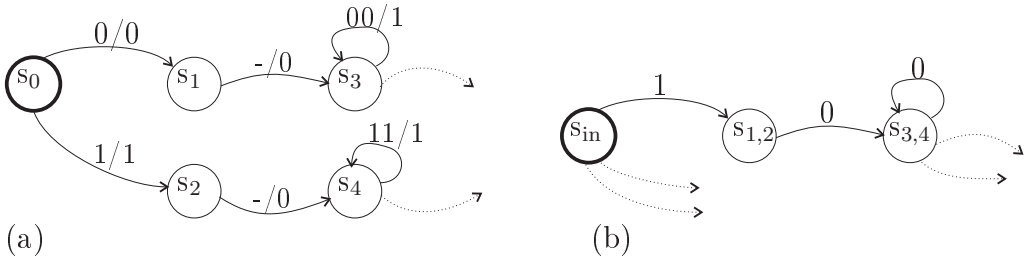


FIGURE 3.5 – Début d'un codeur à états finis synchronisé bit associé à un code catastrophique (a) et début du graphe des distances entre paires correspondant (b)

3.3 Conclusion

Dans ce chapitre nous avons présenté comment les méthodes de fonction transfert établies sur graphe pour les codes à longueur fixe peuvent être généralisées pour calculer la distance libre des codes à états finis à LV. La méthode résultante est très efficace comparée aux méthodes précédentes (le chapitre 6 présente les résultats expérimentaux) et n'a aucun problème pour traiter les codes catastrophiques.

Chapitre 4

Bornes sur les distances libres des codes conjoints

La méthode de recherche de codes présentée au chapitre 5 repose sur les notions de *code totalement indéterminé*, de *code partiellement déterminé*(ou *code incomplet*) et de *code totalement déterminé* (ou *code complet*) ; ainsi que sur des bornes sur les distances libres de ces codes partiellement ou totalement déterminés.

Le paragraphe 4.1 introduit ces notions de codes partiellement ou totalement déterminés. Ensuite nous faisons un bref rappel sur quelque bornes sur la distance libre existantes dans la littérature au paragraphe 4.2 avant de présenter les bornes que nous proposons au paragraphe 4.3.

4.1 Code partiellement ou totalement déterminé

Cette définition dépend du type de code (code à longueur variable ou code arithmétique).

4.1.1 Code à longueur variable conjoints partiellement ou totalement déterminé

Definition 4.1 *Un code à longueur variable (CLV) conjoint* $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ *est* totalement indéterminé *si aucun des bits* c_i^j *(*$1 \leqslant i \leqslant M$ et $1 \leqslant j \leqslant \ell_i$*) n'est connu ; il est* partiellement déterminé *si quelques* c_i^j *sont connus et il est* totalement

déterminé *si tous les* c_i^j *sont connus.*

Definition 4.2 *Un CLV conjoint partiellement ou totalement déterminé* $\mathcal{C}^1$ *est* déduit *d'un autre CLV conjoint partiellement déterminé* $\mathcal{C}^0$ *(noté* $\mathcal{C}^0 \prec \mathcal{C}^1$*) s'il est obtenu en spécifiant quelques ou tous les bits indéterminés des mots de code de* $\mathcal{C}^0$*.*

Exemple 4.1 *Soit la source* X_3 *à valeur dans l'alphabet* $\mathcal{A}_3$ *auquel on associe le vecteur de Kraft* $\boldsymbol{\ell}_3 = (1, 2, 3)$*. Considérons les dictionnaires* $\mathcal{C}_3^0 = \{c_1^1, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$*,* $\mathcal{C}_3^1 = \{0, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$*,* $\mathcal{C}_3^2 = \{0, 1c_2^2, c_3^1c_3^2c_3^3\}$ *et* $\mathcal{C}_3^3 = \{1, 01, 001\}$*.* $\mathcal{C}_3^0$ *est un dictionnaire totalement indéterminé,* $\mathcal{C}_3^1$ *et* $\mathcal{C}_3^2$ *sont partiellement déterminés et* $\mathcal{C}_3^3$ *est lui totalement déterminé. Par ailleurs, on a* $\mathcal{C}_3^0 \prec \mathcal{C}_3^1$ *puisque,* $\mathcal{C}_3^1$ *peut être obtenu en spécifiant le premier mot de code de* $\mathcal{C}_3^0$*. Le dictionnaire* $\mathcal{C}_3^2$ *peut être obtenu en spécifiant le premier mot de code et le premier bit du deuxième mot de code du dictionnaire* $\mathcal{C}_3^0$*, donc* $\mathcal{C}_3^0 \prec \mathcal{C}_3^2$*. De même, le dictionnaire* $\mathcal{C}_3^3$ *peut être obtenu en spécifiant tous les bits de* $\mathcal{C}_3^0$*, donc* $\mathcal{C}_3^0 \prec \mathcal{C}_3^3$*. On a aussi* $\mathcal{C}_3^1 \prec \mathcal{C}_3^2$ *car* $\mathcal{C}_3^2$ *est obtenu en spécifiant le premier bit du deuxième mot de code de* $\mathcal{C}_3^1$*. Mais* $\mathcal{C}_3^1 \not\prec \mathcal{C}_3^3$ *et* $\mathcal{C}_3^2 \not\prec \mathcal{C}_3^3$ *car le premier mot de code de* $\mathcal{C}_3^3$ *est différent du premier mot de code de* $\mathcal{C}_3^1$ *et de* $\mathcal{C}_3^2$*.*

4.1.2 Code arithmétique conjoint partiellement ou totalement déterminé

Definition 4.3 *Un code arithmétique (CA) conjoint* totalement déterminé *est un code dont le codeur à états finis (CEF) qui décrit son processus de codage est un* automate complet*, c'est-à-dire un automate capable d'encoder toute séquence de la source. Un CA conjoint* partiellement déterminé *est décrit par un* automate incomplet *qui est un automate ayant un ou plusieurs états terminaux sans transitions sortantes, dans lesquels le codage s'arrête.*

La figure 2.9 (a) à la page 52 représente un automate incomplet dans lequel l'état s_1 est un état terminal ; la figure 2.9 (b) représente un automate incomplet dans lequel l'état s_2 est un état terminal et la figure 2.9 (c) représente un automate complet.

Ces états terminaux sont nommés dans la suite *état non exploré*. Un automate incomplet Γ_0 génère des séquences de code préfixes à longueur finie et éventuelle-

ment des séquences de code à longueur infinie. Supposons que le *code partiellement déterminé* $\mathcal{C}_0 = \mathcal{C}(\Gamma_0, s_0)$ contienne ces séquences (finies ou infinies).

Definition 4.4 *La distance libre associée à un CEF incomplet est la distance minimale de Hamming entre toutes les paires de séquence de sortie distinctes, qui sont soit de longueur infinie, soit de longueur égale et associée à des chemins qui convergent dans le même état (tous les chemins partent de l'état s_0). S'il n'existe pas de paires de chemins (de longueur finie) convergeant dans un même état ou (de longueur infinie) qui convergent à un instant donné dans un même état, la distance libre est infinie.*

Definition 4.5 *Un automate incomplet ou complet Γ_1 est* déduit *d'un automate incomplet s'il est obtenu en explorant (en ajoutant les états successeurs) un ou plusieurs états terminaux de Γ_0 (par conséquent, le graphe Γ_0 est un sous-graphe de Γ_1) [GM84]. Soit $\mathcal{C}_0$ et $\mathcal{C}_1$ les deux codes générés respectivement par Γ_0 et Γ_1, alors, $\mathcal{C}_0 \prec \mathcal{C}_1$.*

La figure 4.1 (a) représente un automate incomplet dans lequel l'état s_2 est un état terminal et la figure 4.1 (b) représente un automate complet déduit de l'automate de la figure 4.1 (a).

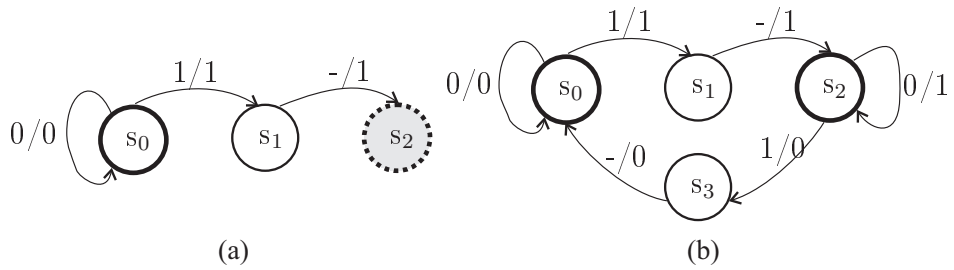


FIGURE 4.1 – Exemple d'un automate incomplet (a) et un automate complet (b) déduit de l'automate (a)

Le fait que $\mathcal{C}_0 \prec \mathcal{C}_1$, implique que tous les éléments dans $\mathcal{C}_0$ sont préfixes d'éléments dans $\mathcal{C}_1$. Soient $d^{(0)}_{\text{libre}}$ et $d^{(1)}_{\text{libre}}$ les distances libres respectives de $\mathcal{C}_0$ et $\mathcal{C}_1$. Si $\mathcal{C}_0 \prec \mathcal{C}_1$, une propriété importante qui suit directement des définitions 2.14 et 4.4 (voir pages 55 et 69 respectivement) est que $d^{(0)}_{\text{libre}}$ est une borne supérieure de $d^{(1)}_{\text{libre}}$.

Lemme 4.1 *Soient $\mathcal{C}_0$ et $\mathcal{C}_1$ deux codes (incomplets) de tel sorte que $\mathcal{C}_0 \prec \mathcal{C}_1$. Alors*

$$d_{\text{libre}}(\mathcal{C}_0) \geqslant d_{\text{libre}}(\mathcal{C}_1). \tag{4.1}$$

Par conséquent, lors du processus de déduction de code, la distance libre ne peut que décroitre ou rester la même.

4.2 Quelques bornes existantes

Ces bornes sont issues des codes de canal en bloc ou convolutifs.

4.2.1 Borne de Plotkin

Considérons un code en bloc (non)linéaire $C(n, M, d)$, où n est la longueur de bloc, M est le nombre de mots de code, et d est la distance de Hamming minimale du code. Pour $M > 1$, la borne supérieure de Plotkin [Plo60, Gal68] produit

$$d \leqslant \bar{d}^{\mathrm{p}}(n, M) = \left\lfloor \frac{nM}{2(M-1)} \right\rfloor, \tag{4.2}$$

où $\lfloor x \rfloor$ est le plus grand entier inférieur ou égal à x. Pour $M \in \{0, 1\}$, on définit $\bar{d}^{\mathrm{p}}(n, M) = \infty$.

(4.2) montre que la borne de Plotkin ne nécessite pas de connaitre les bits des mots de code, il suffit juste de connaitre le nombre de mots de code et de leur longueur.

4.2.2 Borne de Heller

Heller a utilisé la borne de Plotkin pour fournir une borne supérieure de la distance libre des codes convolutifs [JZ99, Chap. 3.5], en comptant le nombre de séquences qui partent de l'état zéro et mènent dans le même état. La borne de Heller s'étend à des codes treillis non linéaires qui varient dans le temps et qui génèrent un nombre fixe de bits par transitions, mais ne s'étend pas immédiatement aux codes à états finis à longueur variable, qui incluent les CLV conjoints et les CA conjoints. Ceci sera fait au paragraphe 4.3.2.

4.2.3 Bornes de Buttigieg pour les CLVs

Pour un code à longueur variable $\mathcal{C} = \{c_1, c_2, \ldots, c_M\}$ totalement déterminé, [But95] encadre la distance libre (d_{libre}). Pour cela, il introduit :

1. La *distance minimale globale par bloc* d_{b} définie par

$$d_{\mathrm{b}} = \min\left\{d_{\mathrm{H}}(c_i, c_j) : (c_i, c_j) \in \mathcal{C}_M^2, i \neq j \text{ et } \ell_i = \ell_j\right\}. \tag{4.3}$$

2. La *distance minimale divergente* d_{d} définie par

$$d_{\mathrm{d}} = \min\left\{D_{\mathrm{d}}(c_i, c_j) : (c_i, c_j) \in \mathcal{C}_M^2, \ell_j < \ell_i\right\}, \tag{4.4}$$

   c'est-à-dire la valeur minimale de toutes les distances divergentes entre toutes les paires possibles de mots de code de longueurs différentes. Pour deux mots de code c_i et c_j tel que $l_j < l_i$, la *distance divergente* est définie comme $D_{\mathrm{d}}(c_i, c_j) = d_{\mathrm{H}}(c_i^1 \cdots c_i^{\ell_j}, c_j^1 \cdots c_j^{\ell_j})$.

3. La *distance minimale convergente*, d_{c} définie par :

$$d_{\mathrm{c}} = \min\left\{D_{\mathrm{c}}(c_i, c_j) : (c_i, c_j) \in \mathcal{C}_M^2, \ell_j < \ell_i\right\}, \tag{4.5}$$

   qui est la valeur minimale de toutes les distances convergentes entre toutes les paires possibles de mots de code de longueurs différentes. La *distance convergente* entre deux mots de code $(c_i, c_j) \in \mathcal{C}_M^2$ tel que $l_j < l_i$ est définie comme : $D_{\mathrm{c}}(c_i, c_j) = d_{\mathrm{H}}(c_i^{\ell_i - \ell j + 1} \cdots c_i^{\ell_i}, c_j^1 \cdots c_j^{\ell_j})$.

Cela lui permet d'obtenir l'encadrement suivant

$$\min(d_{\mathrm{b}}, d_{\mathrm{d}} + d_{\mathrm{c}}) \leqslant d_{\text{libre}} \leqslant d_{\mathrm{b}}. \tag{4.6}$$

Si $\min(d_{\mathrm{b}}, d_{\mathrm{d}} + d_{\mathrm{c}}) = d_{\mathrm{b}}$, alors on a la distance libre exacte. Pour les codes où tous les bits ne sont pas spécifiés, cette borne n'est pas directement applicable.

4.3 Bornes sur la distance libre des codes conjoints

Ce paragraphe présente quelques bornes sur la distance libre des codes conjoints.

4.3.1 Application de la borne de Plotkin aux CLV conjoints

Si on ne connait que le vecteur de Kraft $\boldsymbol{\ell}$ (voir page 39), on peut appliquer la borne de Plotkin sur la distance minimale par bloc d_{b} (4.3) pour obtenir une borne supérieure de la distance libre. Soit L la valeur maximale de $\boldsymbol{\ell}$ et M_r pour $1 \leqslant r \leqslant L$

la *multiplicité* des mots de code de longueur r, c'est-à-dire le nombre d'éléments dans $\boldsymbol{\ell}$ égaux à r. On applique (4.2) aux codes en bloc $C_r(r, M_r, d_r)$, $(1 \leqslant r \leqslant L)$, ce qui permet d'obtenir une borne pour chaque longueur de mot de code. Une extension de la borne de Plotkin est alors

$$\bar{d}^{\mathrm{pe}}_{\mathrm{libre}}(\boldsymbol{\ell}) = \min_{1 \leqslant r \leqslant L} \bar{d}^{\mathrm{p}}(r, M_r), \tag{4.7}$$

et on a

$$d_{\mathrm{libre}} \leqslant \bar{d}^{\mathrm{pe}}_{\mathrm{libre}}(\boldsymbol{\ell}). \tag{4.8}$$

Exemple 4.2 *Soit le vecteur de Kraft* $\boldsymbol{\ell} = (2, 3, 5, 5)$. *On a* $M_0 = M_1 = M_4 = 0$, $M_2 = M_3 = 1$, *et* $M_5 = 2$. *L'application de l'extension de la borne de Plotkin à* $\boldsymbol{\ell}$ *conduit à* $\bar{d}^{pe}(\boldsymbol{\ell}) = 5$, *alors qu'on peut montrer que la distance libre maximale réalisable avec* $\boldsymbol{\ell}$ *est* $d_{\mathrm{libre}} = 2$.

Bien que cette méthode permet d'avoir une borne supérieure de la distance libre en n'utilisant que le vecteur de Kraft, la borne est non triviale que s'il existe plusieurs mots de code de longueur égale, et elle sera en général faible puisqu'elle ne prend pas en compte les séquences de mots de code de longueur supérieure à L, c'est-à-dire des séquences obtenues par concaténation de mots de code.

4.3.2 Extension de la borne de Heller

La borne supérieure de Heller sur la distance libre des codes convolutifs repose sur le décompte des séquences de code de longueur n. Ce nombre est ensuite inséré dans la borne de Plotkin [JZ99, Chap. 3.5]. Ce paragraphe étend l'approche de Heller aux codes à états finis à LVs qui incluent les CLV conjoints et CA conjoint.

Soit $\Gamma(\mathcal{S}, \mathcal{T})$ le codeur à états finis à longueur variable associé au codes à longueur variable avec S états. On pose $\mathcal{S} = \{0, \ldots, S-1\}$ sans perte de généralité. Seule la longueur en bits de sortie de chaque transition a besoin d'être connue pour compter le nombre de séquences de code de longueur n. Soit L la longueur maximale en bits de sortie des transitions dans $\mathcal{T}$:

$$L = \max_{u \in \mathcal{T}} \ell(O(u)). \tag{4.9}$$

Pour les CLVs conjoints $S = 1$ et $L = \max(\ell_1, \ell_2, \ldots, \ell_M)$.

Considérons les matrices à valeurs entières $A_r \in \mathbb{N}^{S\times S}$, $1 \leqslant r \leqslant L$ qui comptent le nombre de transitions de longueur r,

$$(A_r)_{i,j} = |\{u : \ell(O(u)) = r \text{ et } \sigma(u) = i \text{ et } \tau(u) = j\}| . \tag{4.10}$$

L'élément $(A_r)_{i,j}$ compte le nombre de transitions entre l'état i et l'état j avec une longueur en bits de sortie égale à r. Pour un CLV conjoint, A_r se réduit à un scalaire qui représente le nombre de mots de code de longueur r.

Le vecteur ligne $\mathbf{m}_n \in \mathbb{N}^S$ contient les entrées $m_{n,s}$ qui représentent le nombre de séquences de longueur n qui ont comme état d'arrivée s, $s = 0, \ldots, S-1$. Pour $n \geqslant 1$, le *vecteur de comptage* $\mathbf{m}_n$ peut être exprimé comme une fonction des vecteurs de comptage précédents, $\mathbf{m}_n = \sum_{r=1}^{L} \mathbf{m}_{n-r} A_r$, ce qui peut être écrit comme

$$\mathbf{m}_n = [\mathbf{m}_{n-L}, \ldots, \mathbf{m}_{n-1}] \left[A_L^{\mathsf{T}} \ldots A_1^{\mathsf{T}}\right]^{\mathsf{T}} . \tag{4.11}$$

La récursion est initialisé avec le vecteur $[\mathbf{m}_{-L+1}, \ldots, \mathbf{m}_0] = [\mathbf{0}, \ldots, \mathbf{0}, \mathbf{e}_i]$, avec $m_{0,i} = 1$ pour l'état initial i et toutes les autres entrées égales à zéro. Maintenant, (4.11) peut être étendu pour calculer un *bloc* de L vecteurs de comptage $[\mathbf{m}_n, \ldots, \mathbf{m}_{n+L-1}]$ en se servant du précédent bloc de L vecteurs de comptage, permettant ainsi de suivre le comportement sur L-bloc. On obtient

$$[\mathbf{m}_n, \ldots, \mathbf{m}_{n+L-1}] = [\mathbf{m}_{n-L}, \ldots, \mathbf{m}_{n-1}] \cdot A, \tag{4.12}$$

où A est une matrice de taille $LS \times LS$ composée de $LS \times S$ blocs $A^{(k)}$,

$$A = \left[A^{(0)}, \ldots, A^{(L-1)}\right] , \tag{4.13}$$

où

$$A^{(k)} = \sum_{j=0}^{k} A_{\to(k-j)} B_j .$$

Cette dernière expression est calculée à partir du bloc $LS\times S$ de départ, $[A_L^{\mathsf{T}}, \ldots, A_1^{\mathsf{T}}]^{\mathsf{T}}$, décalé de iS positions vers le bas,

$$A_{\to(i)} = [\underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{i \text{ fois}}, \; A_L^{\mathsf{T}}, \ldots, A_{1+i}^{\mathsf{T}}]^{\mathsf{T}},$$

où $\mathbf{0}$ représente un bloc de $S \times S$ zéros. Elle implique également la somme de tous les produits de A_r, comme la somme des indices de $j = 1, \ldots, L-1$, donnée par

$$B_j = \sum_{\substack{(r_1,\ldots,r_i):\\ r_1+\ldots+r_i=j}} A_{r_1} \cdot \ldots \cdot A_{r_i} . \tag{4.14}$$

La somme (4.14) est sur toutes les partitions *ordonnées* de j en i entiers non-négatifs $r_1, \ldots, r_i$ (noter que $A_{r_k} = 0$ s'il n'existe pas de mots de code de longueur r_k). L'ordre est important, puisque la multiplication matricielle n'est pas commutative en général. Par définition, $B_0 = I_S$, la matrice identité de taille $S \times S$.

En utilisant (4.12), on peut calculer tous les décomptes à partir d'un état de départ i de manière itérative comme :

$$\left[\mathbf{m}_{(k-1)L+1}(i), \ldots, \mathbf{m}_{kL}(i)\right] = [\mathbf{0}, \ldots, \mathbf{0}, \mathbf{e}_i]\, A^k, \qquad k = 1, 2, \ldots \tag{4.15}$$

Le nombre de séquences de longueur n qui partent de l'état i à l'état j peut être extrait directement de la matrice A^k comme

$$m_{n,j}(i) = \left(A^k\right)_{(L-1)S+i,\ rS+j}, \tag{4.16}$$

où $r = (n-1) \bmod L$ et $k = (n-1-r)/L+1$. Pour les CLVs conjoints, ceci devient $m_n = \left(A^k\right)_{L-1,\ r}$ (les indices commencent à 0).

Le décompte obtenu peut être inséré dans la borne de Plotkin pour produire une borne supérieure sur la distance libre.

Proposition 4.2 *Une extension de la borne supérieure de Heller sur la distance libre des codes à états finis à LVs est*

$$\bar{d}^{\text{he}}_{\text{libre}}(\boldsymbol{\ell}) = \min_{0<n\leqslant n_{\max}} \min_{i,j\in\mathcal{S}} \bar{d}^{\text{p}}\left(n, m_{n,j}(i)\right), \tag{4.17}$$

où $n_{\max} \geqslant L$ est choisi en fonction des contraintes temps de calcul.

Preuve: L'ensemble des chemins de longueur n entre un certain état de départ i et un certain état d'arrivée j forme un code en bloc (non-linéaire), ainsi la borne de Plotkin peut être appliquée. Il reste à montrer que (4.15) compte correctement le nombre de chemins $m_{n,j}(i)$, c'est-à-dire qu'aucun chemin n'est compté deux fois. C'est effectivement le cas dans (4.12), puisque la définition de $A_{\to(i)}$ assure que la transition conduit du bloc $\left[\mathbf{m}_{(k-2)L+1}(i), \ldots, \mathbf{m}_{(k-1)L}(i)\right]$ au bloc $\left[\mathbf{m}_{(k-1)L+1}(i), \ldots, \mathbf{m}_{kL}(i)\right]$, alors que B_j prend en compte les transitions au sein du dernier bloc. □

Cette technique peut être utilisée pour obtenir des bornes sur la distance minimale de schémas pratiques transmettant des blocs de longueur finie, lorsque l'état

initial et la longueur n sont connus par le décodeur. S'il n'existe aucun mécanisme de terminaison, le décodeur ne connait pas l'états final, ainsi la borne devient $\min_{j\in\mathcal{S}} \bar{d}^{\mathrm{p}}(n, m_{n,j}(i))$. Si les blocs sont terminés dans l'état j, la borne devient $\bar{d}^{\mathrm{p}}(n, m_{n,j}(i))$. Ceci montre l'importance d'un mécanisme de terminaison approprié afin d'empêcher la distance minimale de bloc d'être inférieure à la distance libre

Exemple 4.3 *Soit le CLV conjoint avec le vecteur de Kraft de l'exemple 4.2 ($\boldsymbol{\ell} = (2,3,5,5)$) et $n_{\max} = 5$.*

Étant donné que le graphe du CLV dans le cas le plus simple ne contient qu'un seul état, pour trouver le nombre de séquences de même longueur n qui partent du même état et qui arrivent dans le même état, il suffit de compter le nombre de séquences de mots de code de longueur n.

Pour le vecteur de Kraft donné, le nombre de séquence de mots de code de longueur 1 qui partent de l'état initial est $m_1 = 0$, ensuite on a $m_2 = m_3 = 1$.

Le nombre de séquences de mots de code de longueur 4 est $m_4 = 1$. Cette séquence est obtenue par le premier mot de code qui se concatène à lui-même (c_1c_1).

Le nombre de séquence de mots de code qui partent de l'état initial et de longueur 5 est $m_5 = 4$. Ce nombre m_5 correspond aux deux derniers mots de code de longueur 5, à la séquence obtenue en concaténant le premier mot de code et le deuxième mot de code (c_1c_2) et la séquence c_2c_1.

An appliquant la borne de Plotkin à m_1, m_2, m_3, m_4 et m_5, on obtient $\bar{d}^{\mathrm{he}}_{\mathrm{libre}} = 3$ (pour $n = 5$), ce qui est plus proche de l'optimum réalisable $d_{\mathrm{libre}} = 2$ que la borne de l'exemple 4.2.

L'extension de la borne de Heller (4.17) prend en compte des séquences de mots de code et peut donc améliorer la borne de Plotkin (4.7). Cependant, elle ne prend pas en compte la condition de préfixe imposée ici. Les bornes de Plotkin et de Heller sont donc conservatives.

4.3.3 Utilisation du graphe des distances

Pour un vecteur de Kraft donné, on peut construire un dictionnaire totalement indéterminé $\mathcal{C}^0$ dans lequel tous les bits ont des étiquettes symboliques c_i^j (voir paragraphe 2.2.1). Les arcs orientés du graphe des distances entre paires (GDP)

obtenus à partir de $\mathcal{C}^0$ seront étiquetés avec la somme modulo 2 d'étiquettes symboliques. Le GDP d'un dictionnaire partiellement déterminé $\mathcal{C}^1$ déduit de $\mathcal{C}^0$ c'est-à-dire ($\mathcal{C}^0 \prec \mathcal{C}^1$) aura la même structure que le GDP de $\mathcal{C}^0$, avec la différence que certaines (ou toutes) étiquettes c_i^j sont déterminées à être 0 ou 1.

Exemple 4.4 *$\mathcal{C}^1 = \{0, 10, 111\}$ est un dictionnaire totalement déterminé dont le GDP est représenté par la figure 4.2.*

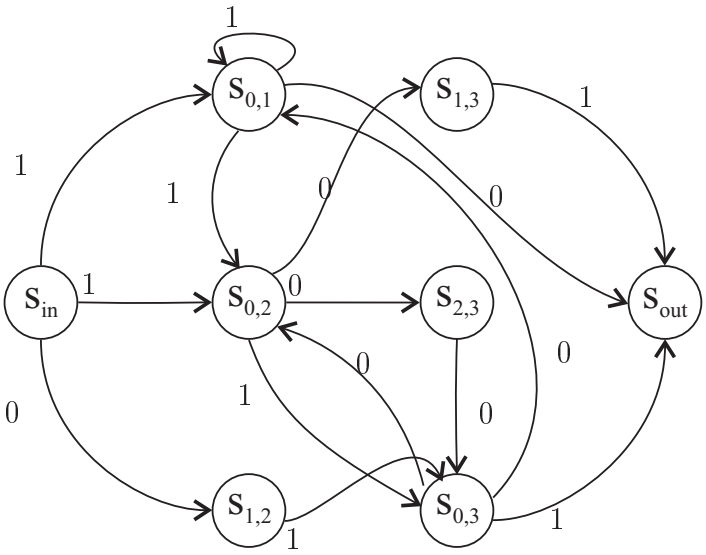


FIGURE 4.2 – GDP de $\mathcal{C}^0 = \{0, 10, 111\}$ dont le codeur à états finis synchronisé bits est représenté sur la figure 2.10 (b) à la page 54

$\mathcal{C}^0 = \{c_1^1, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$ *est un dictionnaire totalement indéterminé dont le codeur à états finis synchronisé symbole est représenté par la figure 4.3 (a). Son codeur à états finis synchronisé bit est représenté par la figure 4.3 (b) et son GDP est représenté par la figure 4.3 (c) (voir paragraphe 3.1 page 57 pour les détails de la construction de GDP à partir du codeur à états finis synchronisé bit).*

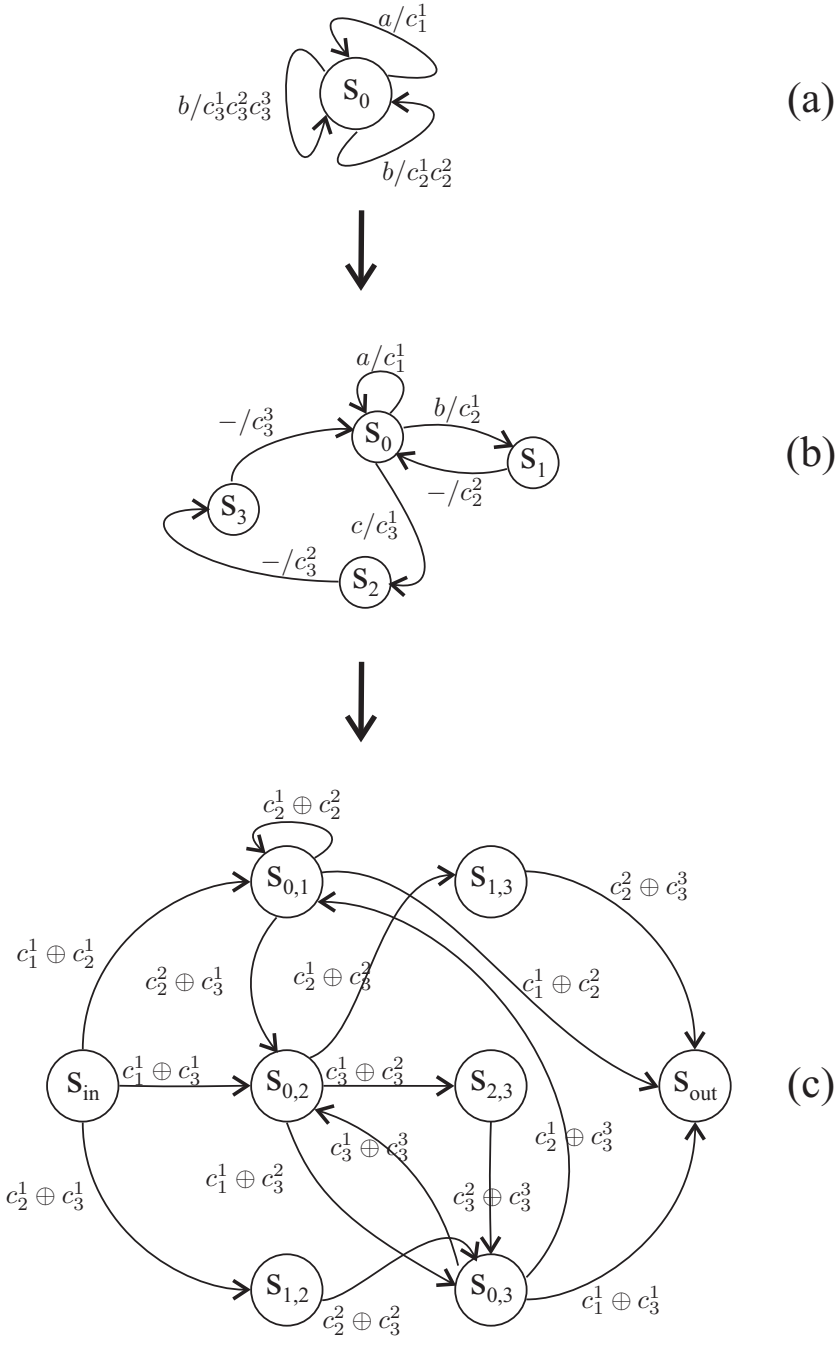


FIGURE 4.3 – Codeur à états finis synchronisé symbole (a), codeur à états finis synchronisé bit (b) et GDP du code $\mathcal{C}^0 = \{c_1^1, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$

Le fait que tous les dictionnaires qui dérivent de $\mathcal{C}^0$ partagent la même structure du GDP permet d'obtenir une hiérarchie imbriquée de bornes supérieures et inférieures sur la distance libre.

Théorème 4.3 *Le poids du chemin de plus petit poids de Hamming obtenu après avoir remplacé l'étiquette de chaque arc indéterminé dans le GDP d'un dictionnaire partiellement déterminé $\mathcal{C}^0$ par 0 (respectivement 1) est une borne inférieure $\underline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0)$ (respectivement une borne supérieure $\overline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0)$) sur les distances libres de tous les CLV conjoints déduits de $\mathcal{C}^0$. Ce chemin peut être trouvé en utilisant l'algorithme de Dijkstra.*

Preuve: La preuve est pour la borne inférieure, celle de la borne supérieure suit les mêmes lignes. Soit $\boldsymbol{e}_{\min}$ un chemin de poids minimal de l'état s_{in} à s_{out} dans le GDP de $\mathcal{C}^0$, lorsque les étiquettes indéterminées sont remplacées par 0. Soit $d_{\text{libre}}^{\min} = w_{\text{H}}(\boldsymbol{e}_{\min})$. Considérons un dictionnaire déduit $\mathcal{C}^1 \succ \mathcal{C}^0$, pour lequel plus d'étiquettes des arcs sur le GDP sont spécifiées comparé au GDP de $\mathcal{C}^0$. Puisque les étiquettes des arcs sont non-négatives, le poids du chemin $\boldsymbol{e}_{\min}$ ne peut que rester le même ou augmenter. Par conséquent, $d_{\text{libre}}^{\min}$ est une borne inférieure sur la distance libre de $\mathcal{C}^1$.

□

Une conséquence directe du théorème 4.3 est que les bornes sur la distance libre du dictionnaire déduit $\mathcal{C}^1$ seront au moins aussi serrées que celles du dictionnaire original $\mathcal{C}^0 \prec \mathcal{C}^1$.

Corollaire 4.4 *Si deux dictionnaires $\mathcal{C}^0$ et $\mathcal{C}^1$ satisfont $\mathcal{C}^0 \prec \mathcal{C}^1$, alors*

$$\left[\underline{d}_{\text{libre}}^{GDP}(\mathcal{C}^1), \overline{d}_{\text{libre}}^{GDP}(\mathcal{C}^1)\right] \subseteq \left[\underline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0), \overline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0)\right]. \tag{4.18}$$

Plus il aura de bits spécifiés dans le dictionnaire partiellement déterminé, plus les bornes fournies par le corollaire 4.4 seront proches les unes des autres.

Exemple 4.5 *Soit le code totalement indéterminé $\mathcal{C}^0 = \{c_1^1, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$ dont le GDP est représentée dans la figure 4.3. En remplaçant tous les arcs indéterminés de la figure 4.3 par 0, on obtient $\underline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0) = 0$ puisque tous les chemins entre s_{in} et s_{out} sont de poids nul. En remplaçant tous les arcs indéterminés de la figure 4.3 par 1, le chemin $(s_{in}, s_{0,1}, s_{out})$ donne $\overline{d}_{\text{libre}}^{GDP}(\mathcal{C}^0) = 2$.*

Quel que soit $\mathcal{C}^1$ qui dérive de $\mathcal{C}^0$, l'application du théorème 4.3 à $\mathcal{C}^0$ conduit à

$$\left[\underline{d}_{\text{libre}}^{GDP}(\mathcal{C}^1), \overline{d}_{\text{libre}}^{GDP}(\mathcal{C}^1)\right] = [0, 2]. \tag{4.19}$$

Ainsi, la distance libre de n'importe quel code déduit de $\mathcal{C}^0$ aura une borne supérieure égale à 2 au maximum.

4.3.4 Bornes sur la distance libre des CLV conjoints préfixes

Les méthodes décrites ci-dessus pour obtenir des bornes sur la distance libre ne tiennent pas en compte du fait qu'aucun mot de code ne peut être préfixe d'un autre. Tout dictionnaire d'un CLV conjoint préfixe peut être représenté par un arbre binaire étiqueté dont les feuilles correspondent aux M symboles sources, de tel sorte que le mot de code d'un symbole peut être lu comme la concaténation des étiquettes binaires de la racine à la feuille correspondante, voir [CT91].

Considérons l'ensemble $\Theta(\boldsymbol{\ell})$ des arbres binaires non étiquetés qui peuvent être associés à des dictionnaires des CLV conjoints préfixes avec le vecteur de Kraft $\boldsymbol{\ell}$, c'est-à-dire l'ensemble des arbres binaires avec M feuilles à des profondeurs $\ell_1, \ell_2, \ldots, \ell_M$. Ceci correspond à l'ensemble des arbres dits canoniques qui seront construits dans le paragraphe 5.3.1.3. La structure de l'arbre $T \in \Theta(\ell)$ détermine quels mots de code ont des préfixes communs ainsi que la longueur de ces préfixes.

La distance libre de n'importe quel dictionnaire d'un CLV conjoint possédant l'arbre T comme représentation peut être bornée directement à partir de la structure de l'arbre, sans spécifier aucun bit de code. Pour ceci, on commence par numéroter les noeuds intermédiaires de l'arbre, y compris la racine. Ensuite, pour chaque noeud intermédiaire i, on étiquette une branche quittant le noeud par b_i et l'autre branche (si elle existe) par $\bar{b}_i$. Cet étiquetage reflète la condition de préfixe de tout dictionnaire $\mathcal{C}$ d'un CLV qui peut être obtenu en étiquetant l'arbre T. Maintenant, lors de la construction du GDP pour borner la distance libre de $\mathcal{C}$, la connaissance de $d_{\text{H}}(b_i, \bar{b}_i) = b_i \oplus \bar{b}_i = 1$ et $d_{\text{H}}(b_i, b_i) = 0$ peut être appliquée aux arcs du GDP étiquetés $b_i \oplus \bar{b}_i$, tandis que toutes les autres étiquettes sont remplacées par 0 ou 1, en fonction du type de borne qui est calculée (voir paragraphe 4.3.3). Les bornes de la distance libre qui en résultent seront désignées par $\underline{d}_{\text{libre}}^{\text{T}}(T)$ et $\overline{d}_{\text{libre}}^{\text{T}}(T)$.

Exemple 4.6 *Considérons une fois de plus le code indéterminé* $\mathcal{C}^0 = \{c_1^1, c_2^1c_2^2, c_3^1c_3^2c_3^3\}$. *Son GDP représenté dans la figure 4.3 ne reflète pas la condition de préfixe. En prenant en compte la condition, on obtiendrait, par exemple,* $\mathcal{C}^1 = \{c_1^1, \bar{c}_1^1c_2^2, \bar{c}_1^1\bar{c}_2^1c_3^3\}$. *Le GDP résultant est représentée dans la figure 4.4. En appliquant le théorème 4.3 à* $\mathcal{C}^1$ *on obtient* $[\underline{d}^T_{\text{libre}}(\mathcal{C}^1), \overline{d}^T_{\text{libre}}(\mathcal{C}^1)] = [1, 2]$ *qui est meilleure que les bornes fournies par* $\underline{d}^{GDP}_{\text{libre}}$.

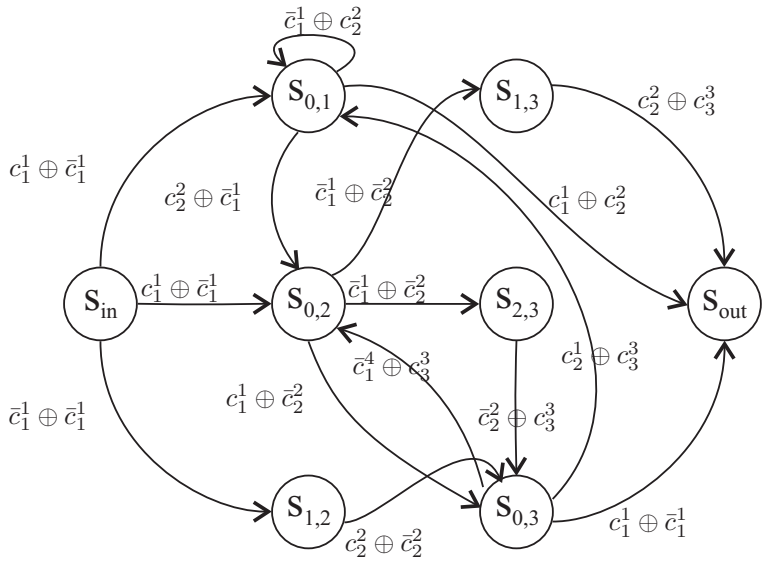


FIGURE 4.4 – GDP déduit du codeur à états finis synchronisé bits de $\mathcal{C}^1 = \{c_1^1, \bar{c}_1^1c_2^2, \bar{c}_1^1\bar{c}_2^1c_3^3\}$, en prenant en compte la condition de préfixe

4.4 Conclusion

Dans ce chapitre, nous avons rappelé quelques bornes existantes sur la distance minimale des codes en bloc (borne de Plotkin) et sur la distances libre des codes convolutifs (borne de Heller).

Nous avons étendu la borne de Plotkin pour proposer une borne supérieure sur la distance libre des codes à longueur variable conjoints (CLV conjoints). Ensuite, nous avons étendu la borne de Heller à l'ensemble des codes à états finis à longueur variable (les codes à états finis à longueur fixe sont un cas particulier des codes à états finis à longueur variable).

Des bornes inférieures et supérieurs de la distance libre des CLV conjoints ont été proposées à partir du graphe des distances entre paires et de la structure en arbre de ces codes CLV conjoints.

Nous utiliserons l'ensemble de ces bornes dans le chapitre 5 pour de la construction des codes source-canal conjoints robustes.

Chapitre 5

Structurer et explorer l'espace des codes

Dans ce chapitre, nous allons présenter les différentes méthodes proposées pour la recherche de bons codes conjoints. D'abord le paragraphe 5.1 présente notre approche générale de construction. Ensuite, le paragraphe 5.3 décrit l'application de cette approche aux CLVs conjoints et aux codes arithmétiques conjoints.

5.1 Espace des codes conjoints

Ce paragraphe décrit la méthode que nous proposons pour chercher des codes conjoints correcteurs d'erreurs avec une distance libre maximale. Notre approche consiste à explorer deux sous-ensembles de l'ensemble $\mathcal{F}$ de tous les CEFs. $\mathcal{F}$ contient l'ensemble $\mathcal{F}_{\text{CEF-CLV}}$ de tous les CEFs correspondant aux CLVs conjoints et $\mathcal{F}_{\text{CEF-CA}}$ de tous les CEFs correspondant aux CAs conjoints (voir figure 5.1). Dorénavant $\mathcal{F}_{\text{CSCC}}$ (avec CSCC pour code source-canal conjoint) représente soit $\mathcal{F}_{\text{CEF-CLV}}$ soit $\mathcal{F}_{\text{CEF-CA}}$.

$\mathcal{F}_{\text{CSCC}}$ quant à lui contient l'ensemble $\mathcal{F}^{\text{T}}_{\text{CSCC}}$ de tous les CEFs représentant un CLV/CA suivis d'un code correcteur d'erreurs c'est-à-dire la séparation classique du codage tandem). Il contient aussi l'ensemble $\mathcal{F}^{\text{M}}_{\text{CSCC}}$ de tous les CLV/CA conjoints *sans mémoire*, c'est-à-dire les codes à états finis conjoints dont le comportement dépend seulement de l'état courant du codeur. Dans le codeur arithmétique, le comportement du codeur peut dépendre des symboles précédemment codés ou des bits

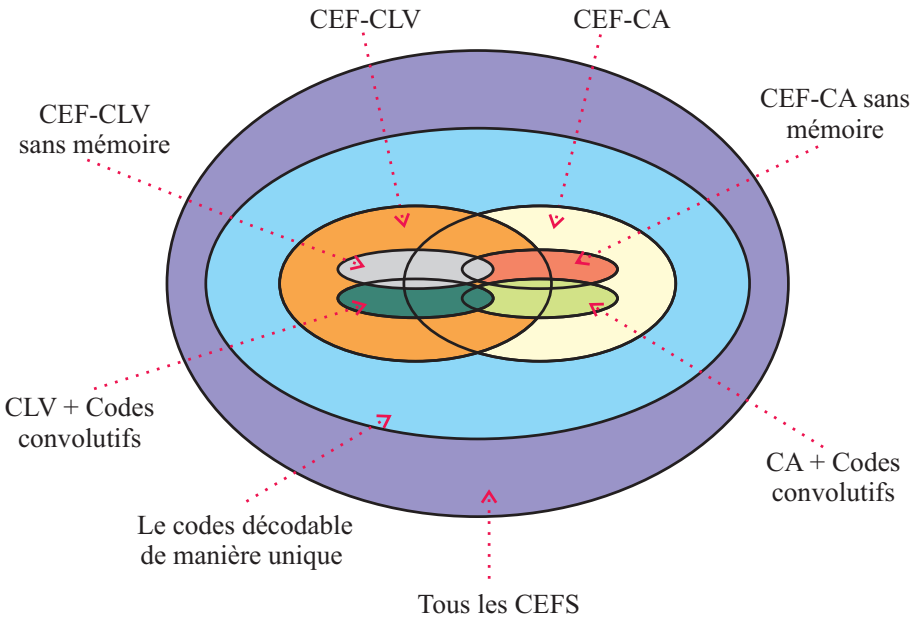


FIGURE 5.1 – Représentation des ensembles des CEFs

de sortie précédemment générés seulement indirectement à travers l'état du codeur. Des CLVs/CAs suivis de code en bloc appartiennent à l'ensemble $\mathcal{F}^{\mathrm{M}}_{\mathrm{CSCC}}$, mais plus généralement, les schémas tandem composé de CLV/CA suivi de codes convolutifs ne font pas parti de l'ensemble $\mathcal{F}^{\mathrm{M}}_{\mathrm{CSCC}}$. Pour des raisons de simplicité, on restreint notre exploration à l'ensemble $\mathcal{F}^{\mathrm{M}}_{\mathrm{CSCC}}$.

Comme l'ensemble $\mathcal{F}^{\mathrm{M}}_{\mathrm{CSCC}}$ reste toujours assez grand, il serait intéressant de le structurer de telle sorte qu'il puisse être exploré efficacement pour trouver le code de plus grande distance libre. Ceci peut être fait à travers un *arbre de codeurs à états finis* dans lequel les feuilles correspondent aux CLVs totalement déterminés (ou aux automates complets pour les CAs) et les noeuds intermédiaires correspondent aux CLVs partiellement déterminés (ou aux automates incomplets pour les CAs). A partir de la racine qui est le CLV totalement indéterminé (ou l'automate incomplet initial déterminé par les paramètres initiaux du codeur pour les CAs) l'arbre est généré en explorant successivement les CLVs partiellement déterminés intermédiaires (ou les automates incomplets intermédiaires pour les CAs) comme décrit dans les paragraphes 5.3.1 et 5.3.2.

En utilisant (4.18) du corollaire 4.4 pour les CLVs (ou le lemme 4.1 pour les CAs) dans un algorithme de *branch-and-prune*, on réduit substantiellement la taille de l'espace de recherche nécessaire pour trouver le meilleur code conjoint concernant

la distance libre.

5.2 Algorithme générique de recherche de bons codes

L'idée de l'algorithme *branch-and-prune* est d'éliminer successivement de grandes parties de l'arbre de codeurs à états finis qui ne peuvent pas conduire à la distance libre optimale. Ceci est fait en mettant à jour itérativement une borne inférieure $\underline{d}_{\text{libre}}$, de la plus grande distance libre qui peut être obtenue. Lorsqu'on explore l'arbre, pour tout code incomplet, la borne supérieure de sa distance libre est comparée à $\underline{d}_{\text{libre}}$. Si elle est plus grande, le code incomplet est étendu, sinon, le code n'est plus exploré puisque selon (4.18) (pour les CLVs) et lemme 4.1 (pour les CAs), tout code complet ou incomplet qui dérive de lui aura une distance libre inférieure ou égale à $\underline{d}_{\text{libre}}$. Si un code complet est atteint et sa distance libre est supérieure à $\underline{d}_{\text{libre}}$, alors $\underline{d}_{\text{libre}}$ est mis à jour.

Trois méthodes d'exploration de l'arbre sont considérées : *exploration en profondeur*, *exploration en largeur*, et la *méthode par tri* qui étend l'automate incomplet avec la plus grande borne supérieure de la distance libre. Leur efficacité respectives sont comparées au chapitre 6.

L'algorithme fourni dans le tableau 5.1 réalise une exploration de l'arbre des codeurs à états finis avec un algorithme de type *branch-and-prune* générique pour obtenir un code $\mathcal{C}_{\text{opt}}$ avec des propriétés de distance optimales en commençant par un code incomplet $\mathcal{C}_0$.

A l'étape 1, la liste de travail $\mathcal{L}$ est initialisée avec $\mathcal{C}_0$, la liste des solutions temporaires $\mathcal{S}$ est vide, la distance libre de référence $\underline{d}_{\text{libre}}$ est mise à zéro et la borne supérieure courante de la meilleure distance libre est obtenue a partir de $\mathcal{C}_0$ en utilisant une des méthodes décrites dans le paragraphe 4.3 pour le CLV ou la définition 4.4 (à la page 69) pour le CA.

Aux étapes 3 et 4, un code candidat $\mathcal{C}$ est extrait de $\mathcal{L}$ en fonction de la méthode d'exploration de l'arbre. Si $\overline{d}_{\text{libre}}(\mathcal{C}) > \underline{d}_{\text{libre}}$ alors on passe à l'étapes 5 si le code est complet, sinon on passe à l'étape 7. Aux étapes 5 et 6, les codes complets sont insérés dans la liste des solutions temporaires $\mathcal{S}$. A l'étape 8, plusieurs codes déduits

TABLE 5.1 – Algorithme générique d'optimisation de code

$\mathcal{C}_{\text{opt}}$ =Optimiser($\mathcal{C}_0$)

1 $\mathcal{L} = \{\mathcal{C}_0\}$; $\mathcal{S} = \emptyset$; $\underline{d}_{\text{libre}} = 0$;
2 faire
3 prendre $\mathcal{C}$ dans $\mathcal{L}$;
4 si $\overline{d}_{\text{libre}}(\mathcal{C}) > \underline{d}_{\text{libre}}$ alors
5 si $\mathcal{C}$ est complet, alors
6 $\mathcal{S} = \mathcal{S} \cup \mathcal{C}$;
7 sinon
8 $\left\{\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(k)}\right\}$ =Déduire($\mathcal{C}$) ;
9 insérer $\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(k)}$ dans $\mathcal{L}$;
10 $\underline{d}_{\text{libre}} = \max_{\mathcal{C}\in\mathcal{S}} d_{\text{libre}}(\mathcal{C})$;
11 éliminer tout $\mathcal{C} \in \mathcal{L}$ tel que $\overline{d}_{\text{libre}}(\mathcal{C}) < \underline{d}_{\text{libre}}$;
12 tans que $\mathcal{L} \neq \emptyset$
13 $\mathcal{C}_{\text{opt}} = \arg\max_{\mathcal{C}\in\mathcal{S}} d_{\text{libre}}(\mathcal{C})$;fin.

de $\mathcal{C}$ sont construits et stockés dans $\mathcal{L}$ à l'étape 9. A l'étape 10, $\underline{d}_{\text{libre}}$ est mis à jour en choisissant dans la liste des solutions temporaires la meilleur distance libre. L'étape 11 élague tous les codes partiellement déterminés en prenant en compte les résultats du corollaire 4.4 pour les CLVs conjoints et le lemme 4.1 pour les CAs conjoints. La clé d'un élagage efficace est la disponibilité d'une bonne borne supérieure de la distance libre des codes incomplets qui est non croissante lorsqu'on parcourt l'arbre de la racine aux feuilles.

A la fin de l'étape 11 la liste $\mathcal{L}$ est ordonnée en fonction de la méthode d'exploration choisie (en profondeur, en largeur, ou la méthode par tri).

- Dans le cas de l'exploration en profondeur, les codes partiellement déterminés appartenant à la branche la plus à gauche de l'arbre des codeurs à états finis sont explorés d'abord.
- Avec l'exploration en largeur, les codes indéterminés appartenant au même niveau horizontal de l'arbre des codeurs à états finis sont explorés en premiers.
- Avec l'exploration avec la méthode par tri le code avec la plus grande borne supérieure de la distance libre est exploré en premier, puisqu'il a le potentiel de fournir la plus grande distance libre.

Dans le cas des CLVs conjoints, l'efficacité de la méthode par tri peut encore être améliorée en utilisant la borne inférieure de la distance libre comme un critère de tri secondaire, c'est-à-dire pour la même borne supérieure de la distance libre, le dictionnaire avec la plus grande borne inférieure de la distance libre sera considéré en premier (le critère de tri ne doit pas être inversé, puisque la borne supérieure s'avère beaucoup plus discriminante pour les dictionnaires incomplets).

L'objectif de l'algorithme Optimiser du tableau 5.1 est de fournir un bon code conjoint sans explorer en entier l'arbre de recherche en élaguant les noeuds correspondant à des codes avec des mauvaises propriétés de distance.

Maintenant que nous avons décrit comment un arbre de codeurs à états finis structuré peut être parcouru efficacement pour trouver un code avec la meilleur distance libre, nous allons décrire la construction de cet arbre des codeurs à états finis. Dans le cas des CLVs conjoints, il existe plusieurs manière de déduire un dictionnaire d'un dictionnaire donné (étape 8). La règle de déduction conduit à plusieurs organisation de la recherche en arbre, comme détaillé dans le paragraphe 5.3.1. Dans le cas des CAs conjoints, nous avons également défini une manière de déduire un automate d'un automate incomplet donné, voir paragraphe 5.3.2.

5.3 Hiérarchie de codes incomplets

5.3.1 Arbres de codes à longueur variables

L'approche proposée est d'organiser tous les CLVs conjoints pour un vecteur de Kraft donné $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ dans une structure en arbre, de telle sorte que chaque feuille correspond à un dictionnaire CLV conjoint donné, et les noeuds parents (intermédiaires) correspondent à des dictionnaires partiellement déterminés, à partir desquels les noeuds fils (dictionnaires) peuvent être obtenus par des règles spécifiques (à décrire). Cette structure d'arbre de données ne doit pas être confondue avec la représentation en arbre des codes préfixes CLVs.

Dans la suite de ce rapport, on suppose (sans perte de généralité) que le vecteur de Kraft $\boldsymbol{\ell}$ est trié en ordre croissant : $\ell_1 \leqslant \ell_2 \leqslant \cdots \leqslant \ell_M$.

Le choix du vecteur de Kraft conduisant à un code avec le minimum de redondance pour une d_{libre} donnée ou qui maximise la d_{libre} pour une redondance donnée

n'est pas considéré dans cette thèse. Des vecteurs de Kraft satisfaisant certaines conditions peuvent être générés en utilisant les bornes fournies au chapitre 4 et le concept de *vecteurs de longueur dominante* présenté dans [Sav09].

Pour un vecteur de Kraft donné, il existe $2^{\sum_{i=1}^{M} \ell_i}$ dictionnaires possibles, en incluant ceux qui ne sont pas décodables de manière unique. Ici, nous allons considérer seulement les codes préfixes (c'est-à-dire les codes dans lesquels aucun mot de code n'est préfixe d'un autre mot de code). Ces codes sont décodables de manière instantanée.

Le nombre de codes préfixes, N^{pr}, pour un vecteur de Kraft donné $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ peut être obtenue de la manière suivante. Soit N_i^{pr} (pour $1 < i \leqslant M$) le nombre de choix du mot de code de longueur ℓ_i lorsque les mots de codes de longueur ℓ_j pour $1 \leqslant j \leqslant i-1$ sont déjà fixés :

$$N_i^{\text{pr}} = 2^{\ell_i} - \sum_{j=1}^{i-1} 2^{\ell_i - \ell_j} \text{ avec } N_1^{\text{pr}} = 2^{\ell_1}. \tag{5.1}$$

Alors,

$$N^{\text{pr}} = \prod_{i=1}^{M} N_i^{\text{pr}}. \tag{5.2}$$

La formule donnée par (5.1) se justifie de la manière suivante. Il existe 2^{ℓ_i} choix possibles pour le mot de code de longueur ℓ_i sans prendre en compte les mots de code de longueur ℓ_j, $j < i$. Lorsque le mot de longueur ℓ_j, $j < i$, est fixé, pour que le dictionnaire reste préfixe, toutes les séquences de bits de longueur ℓ_i et qui ont pour préfixe les ℓ_j premiers bits le $j^{\text{ième}}$ mot de code, doivent être exclues du choix de $i^{\text{ième}}$ mot de code. Ce nombre de séquences est de $2^{\ell_i - \ell_j}$.

Exemple 5.1 *Considérons le vecteur de Kraft* $\boldsymbol{\ell}_5 = (4, 4, 5, 5, 6)$. *Le nombre de dictionnaire possibles en incluant ceux qui ne sont par préfixes est de*

$$N^{\text{T}} = 2^{\sum_{i=1}^{5} \ell_i} = 2^{24} = 16,777,216 \ \textit{dictionnaires possibles.} \tag{5.3}$$

En considérant que les dictionnaire préfixes, on a :

$$
\begin{aligned}
N_1^{\mathrm{pr}} &= 2^{\ell_1} = 16, \\
N_2^{\mathrm{pr}} &= 2^{\ell_2} - 2^{\ell_2-\ell_1} = 16 - 1 = 15, \\
N_3^{\mathrm{pr}} &= 2^{\ell_3} - 2^{\ell_3-\ell_2} - 2^{\ell_3-\ell_1} = 32 - 2 - 2 = 28, \\
N_4^{\mathrm{pr}} &= 2^{\ell_4} - 2^{\ell_4-\ell_3} - 2^{\ell_4-\ell_2} - 2^{\ell_4-\ell_1} = 32 - 1 - 2 - 2 = 27, \\
N_5^{\mathrm{pr}} &= 2^{\ell_5} - 2^{\ell_5-\ell_4} - 2^{\ell_5-\ell_3} - 2^{\ell_5-\ell_2} - 2^{\ell_6-\ell_1} = 64 - 2 - 2 - 4 - 4 = 52, \\
N^{\mathrm{pr}} &= 16 \times 15 \times 28 \times 27 \times 52 = 9,434,880 \ \textit{dictionnaires préfixes.}
\end{aligned}
$$

Quoi qu'il en soit, l'espace de recherche reste immense pour des tailles de vecteurs de Kraft raisonnables.

Dans la suite, nous présentons trois méthodes pour structurer les arbres des CLVs conjoints, c'est-à-dire pour créer des hiérarchies pour les dictionnaires partiellement déterminés. De plus, certaines propriétés de symétrie seront utilisées pour ne pas représenter dans l'arbre les dictionnaires connus pour avoir les mêmes propriétés de distances que certains dictionnaires déjà considérés.

5.3.1.1 Construction par mot de code

Une première méthode pour structurer l'arbre des dictionnaires introduit dans le paragraphe 5.1 est de spécifier tous les bits d'un mot de code à chaque itération. On commence à la racine de l'arbre avec un dictionnaire totalement indéterminé. Les noeuds fils, représentant les dictionnaires déduits à l'étape 8 de Optimiser (voir tableau 5.1), héritent du dictionnaire partiellement déterminé $\mathcal{C}$ associé à leur noeud père et ont un mot de code déterminé en plus. L'arbre de recherche contient donc M niveaux.

La fonction Déduire à l'étape 8 du tableau 5.1 doit être telle que seuls les dictionnaires déduits qui satisfont la condition de préfixe sont conservés. Pour cela, soit $\mathrm{B}_\ell(x)$, la représentation binaire de $x \in \mathbb{N}$ utilisant $\ell \geq \lceil \log_2(x+1) \rceil$ bits, où $\lceil y \rceil$ représente le plus petit entier supérieur ou égal à y. On a par exemple, $\mathrm{B}_3(1) = 001$. Considérons un dictionnaire partiellement déterminé $\mathcal{C}$ (dans ce paragraphe, si le bit c_i^j est indéterminé dans le mot de code c_i, alors tous les bits dans c_i sont indéterminés), et supposons que $\mathcal{C}^{\mathrm{d}} \subset \mathcal{C}$ contient les mots de codes déterminés de $\mathcal{C}$.

Maintenant

$$\text{pref}(\mathcal{C}, \ell, x) = \begin{cases} 1 & \text{si } \mathcal{C}^{\text{d}} \cup \{\text{B}_\ell(x)\} \text{ est préfixe,} \\ 0 & \text{sinon.} \end{cases} \tag{5.4}$$

Exemple 5.2 *Considérons* $\mathcal{C} = \{1, 01, c_3^1 c_3^2 c_3^3, c_4^1 c_4^2 c_4^3\}$. *Si* $x = 1$, *et* $\ell = 3$, $\mathcal{C}^d \cup \{\text{B}_\ell(x)\} = \{1, 01, 001\}$, *alors* $\text{pref}(\mathcal{C}, 3, 1) = 1$. *Si* $x = 3$ *et* $\ell = 3$, *alors* $\mathcal{C}^d \cup \{\text{B}_\ell(x)\} = \{1, 01, 011\}$ *n'est pas préfixe et* $\text{pref}(\mathcal{C}, 3, 3) = 0$.

La fonction (5.4) permet de construire une instance de la fonction Déduire comme détaillé dans le tableau 5.2. Dans cette fonction, le mot de code indéterminé le plus court dans $\mathcal{C}$ est toujours spécifié d'abord (voir étape 3). Un nouveau dictionnaire est obtenu en remplaçant $\mathbf{c}_{\text{e}}$, le plus court mot de code indéterminé, par $\text{B}_\ell(x)$ seulement si le dictionnaire résultant est potentiellement préfixe (voir les étapes 6 à 8).

TABLE 5.2 – Déduction d'un dictionnaire via la détermination du plus court mot de code indéterminé

$\left\{\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(k)}\right\}$ =Déduction_par_mot de code($\mathcal{C}$)

1	Si $\mathcal{C}$ est entièrement déterminé, renvoyer $\{\mathcal{C}\}$.
2	construire $\mathcal{C}^{\text{d}}$ contenant tous les mots de codes spécifiés de $\mathcal{C}$;
3	$\ell = \min_{\mathbf{c} \in \mathcal{C} \setminus \mathcal{C}^{\text{d}}} \ell(\mathbf{c})$; $\mathbf{c}_{\text{e}} = \arg\min_{\mathbf{c} \in \mathcal{C} \setminus \mathcal{C}^{\text{d}}} \ell(\mathbf{c})$;
4	$k = 0$;
5	pour $x = 0$ à $2^\ell - 1$
6	si $\text{pref}(\mathcal{C}, \ell, x) = 1$
7	$k = k + 1$;
8	$\mathcal{C}^{(k)} = (\mathcal{C} \setminus \{\mathbf{c}_{\text{e}}\}) \cup \text{B}_\ell(x)$;
	fin si
	fin pour
9	renvoyer $\left\{\mathcal{C}^{(1)}, \ldots, \mathcal{C}^{(k)}\right\}$.

Exemple 5.3 *La figure 5.2 montre un arbre de recherche correspondant au vecteur de Kraft* $\boldsymbol{\ell} = \{1, 2, 3\}$ *lorsque la déduction décrite dans le tableau 5.2 est utilisée. Les bits indéterminés sont représentés par* x. *Puisque* $\ell_1 = 1$ *est la plus petite longueur, les fils de la racine (contenant un dictionnaire entièrement indéterminé) correspondent à* 2^{ℓ_1} *dictionnaires contenant chacun un seul mot de code déterminé* $\text{B}_{\ell_1}(x)$

($0 \leqslant x < 2^{\ell_1}$) de ℓ_1 bits. Les codes résultants sont nommés $\mathcal{C}^{(1)}$ à $\mathcal{C}^{(2^{\ell_1})}$. Ensuite pour chaque dictionnaire $\mathcal{C}^{(k)}$, $1 \leqslant k \leqslant 2^{\ell_1}$, le mot de code de longueur $\ell_2 = 2$ est spécifié. Les dictionnaires résultants sont à leur tour étendus avec le mot de code de longueur $\ell_3 = 3$. Par conséquent tous les CLVs conjoints correspondant au vecteur de Kraft $\boldsymbol{\ell} = \{1, 2, 3\}$ sont représentés par les feuilles de l'arbre de recherche.

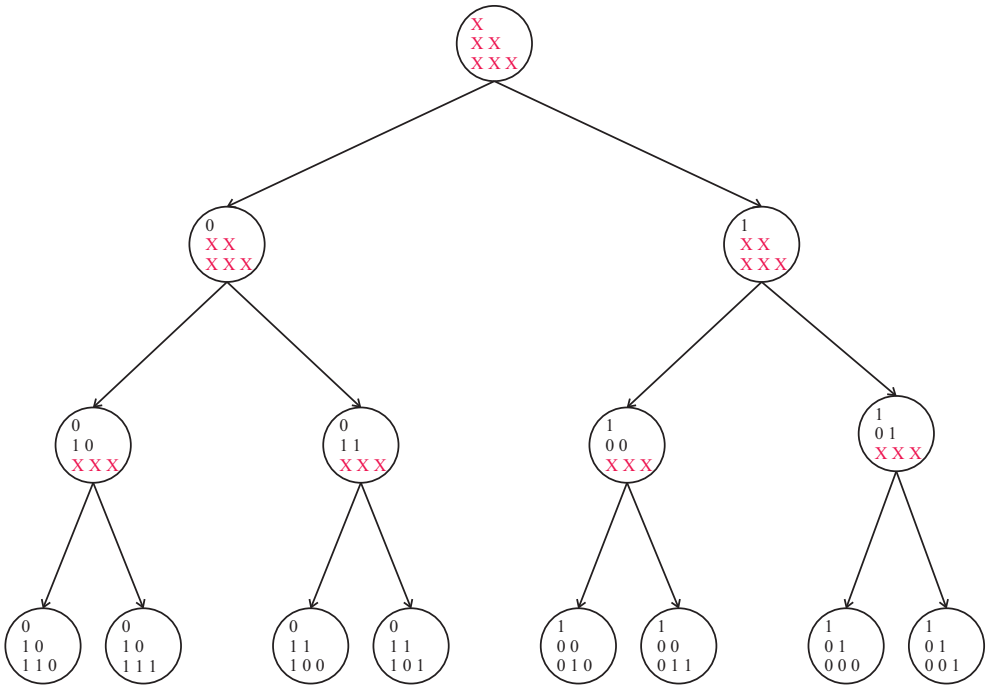


FIGURE 5.2 – Exemple d'un arbre de recherche de CLVs conjoints pour le vecteur Kraft $\boldsymbol{\ell} = (1, 2, 3)$ lorsque les dictionnaires sont construits par mot de code

Inverser tous les bits d'un dictionnaire ne change pas ses propriétés de distance. Pour éviter de traiter un dictionnaire dont les bits sont inversés par rapport à un premier dictionnaire déjà considéré, on peut réaliser la première déduction à partir du noeud racine (lorsque $\mathcal{C}$ dans le tableau 5.1 est totalement indéterminé) avec x allant de 0 à $2^{\ell_1 - 1}$ à l'étape 8 de Optimiser. Cette propriété divise par deux la taille de l'espace de recherche pour trouver le meilleur CLV conjoint sans affecter les propriétés de distance.

Si les longueurs ℓ_i et ℓ_{i+1} de deux mots de codes sont égales, alors, échanger les mots de code i et $i + 1$, ne changera pas la distance libre du dictionnaire correspondant. Par conséquent, on peut imposer un ordre lexicographique lors de la

construction des dictionnaires. Ceci réduit de manière significative la taille de l'espace de recherche pour trouver le meilleur CLV conjoint.

5.3.1.2 Construction par plan de bits

Cette méthode de construction spécifie tous les bits d'un même plan de bits : le premier bit de tous les mots de code est spécifié, ensuite le second bit (quand il est présent), et ainsi de suite. En faisant ceci, on doit s'assurer que les suffixes des mots de codes qui ont le même préfixe satisfont l'inégalité de Kraft (2.9 à la page 39) pour garantir que le dictionnaire global reste préfixe. Soit L est la longueur maximale dans le vecteur de Kraft $\boldsymbol{\ell}$, alors l'arbre de recherche contient L niveaux.

Considérons un dictionnaire partiellement déterminé $\mathcal{C}$. Soit k l'index du premier plan de bits indéterminé et $i \in [1, M]$, l'indexe du mot de code le plus court qui contient encore des bits indéterminés (les codes sont classés par ordre croissant des longueurs). Il existe $2^{(M-i+1)}$ différentes manières de spécifier les bits indéterminés du $k^{\text{ième}}$ plan de bits $\mathcal{C}$. Notons $\mathcal{C}^{(j)}$, $j = 1, \ldots, 2^{(M-i+1)}$ les dictionnaires (in)déterminés déduits de $\mathcal{C}$. Considérons un $\mathcal{C}^{(j)}$, et désignons par $\mathcal{C}^{(j)}_{m,k}$ le $m^{\text{ième}}$ sous-ensemble de $\mathcal{C}^{(j)}$ dont les mots de code ont le même préfixe sur les $k-1$ premiers bits, alors pour que le code $\mathcal{C}$ reste potentiellement préfixe, $\mathcal{C}^{(j)}_{m,k}$ doit satisfaire

$$\sum_{\mathbf{c} \in \mathcal{C}^{(j)}_{m,k}} 2^{-(\ell(\mathbf{c})-k+1)} \leqslant 1, \quad \text{pour } m = 1, \ldots, \nu_j, \tag{5.5}$$

où ν_j est le nombre de sous-ensembles. Cette condition peut réduire le nombre de dictionnaire déduits qui peuvent être encore pris en considération.

Exemple 5.4 *Considérons le vecteur de Kraft* $\boldsymbol{\ell}_6 = (2, 3, 3, 4, 4, 4)$. $\mathcal{C}^{(1)}$ *et* $\mathcal{C}^{(2)}$ *représentés ci-dessous sont deux exemples de dictionnaires partiellement déterminés, qu'on pourrait associer au vecteur de Kraft* $\boldsymbol{\ell}_6$. *Sur ces dictionnaires,* x *représente les bits non déterminés. Les deux premiers plans de bits des dictionnaires sont déjà spécifiés et le plan de bit suivant à déterminer est* $k = 3$. *Le mot le plus court contenant encore des bits indéterminés est* c_2. *En considérant le dictionnaire* $\mathcal{C}^{(1)}$, *on constate que les mots de code* c_2 *et* c_4 *ont le même préfixe* "10" *(*$\mathcal{C}^{(1)}_{1,3} = \{c_2, c_4\}$*), alors que les mots de code* c_3, c_5 *et* c_6 *ont le même préfixe* "01" *(donc* $\mathcal{C}^{(1)}_{2,3} = \{c_3, c_5, c_6\}$*). On*

$$\mathcal{C}^{(1)} = \begin{array}{l} 0\,0 \\ 1\,0\,\mathrm{x} \\ 0\,1\,\mathrm{x} \\ 1\,0\,\mathrm{x}\,\mathrm{x} \\ 0\,1\,\mathrm{x}\,\mathrm{x} \\ 0\,1\,\mathrm{x}\,\mathrm{x} \end{array} \qquad \mathcal{C}^{(2)} = \begin{array}{l} 0\,0 \\ 1\,0\,\mathrm{x} \\ 1\,0\,\mathrm{x} \\ 0\,1\,\mathrm{x}\,\mathrm{x} \\ 1\,0\,\mathrm{x}\,\mathrm{x} \\ 1\,1\,\mathrm{x}\,\mathrm{x} \end{array}$$

a :

$$\begin{aligned} \sum_{\mathbf{c}\in\mathcal{C}^{(1)}_{1,3}} 2^{-(\ell(\mathbf{c})-3+1)} &= 2^{-(\ell_2-3+1)} + 2^{-(\ell_4-3+1)} = 0.75 \\ \sum_{\mathbf{c}\in\mathcal{C}^{(1)}_{2,3}} 2^{-(\ell(\mathbf{c})-3+1)} &= 2^{-(\ell_3-3+1)} + 2^{-(\ell_5-3+1)} + 2^{-(\ell_6-3+1)} = 1. \end{aligned}$$

Donc le dictionnaire $\mathcal{C}^{(1)}$ satisfait (5.5), il serait possible de déduire de $\mathcal{C}^{(1)}$ un dictionnaire totalement déterminé et préfixe. Par contre en considérant le dictionnaire $\mathcal{C}^{(2)}$, on a $\mathcal{C}^{(1)}_{2,3} = \{c_2, c_3, c_5\}$ et :

$$\sum_{\mathbf{c}\in\mathcal{C}^{(2)}_{1,3}} 2^{-(\ell(\mathbf{c})-3+1)} = 2^{-(\ell_3-3+1)} + 2^{-(\ell_4-3+1)} + 2^{-(\ell_5-3+1)} = 1,25.$$

le dictionnaire $\mathcal{C}^{(2)}$ ne satisfait pas (5.5), donc on ne peut pas en déduire un dictionnaire parfaitement déterminé qui soit préfixe, par conséquent $\mathcal{C}^{(2)}$ peut être éliminé.

L'algorithme Déduire correspondant est détaillé dans le tableau 5.3. A l'étape 8, le $k^{\text{ième}}$ bit du $\lambda^{\text{ième}}$ mot de code de $\mathcal{C}$ est assigné au $(\lambda - i)^{\text{ième}}$ bit de j. Par conséquent, $2^{(M-i+1)}$ dictionnaires déduits candidats sont obtenus et seulement ceux pour lesquels (5.5) est satisfait pour tous les sous-dictionnaires avec le même préfixe sont conservés.

La figure 5.3 montre un exemple d'arbre de recherche quand la déduction des dictionnaire est faite par plan de bits. A partir de la racine qui contient le dictionnaire entièrement indéterminé, les noeuds fils sont générés, correspondant à toutes les combinaisons possibles de chaque bit de chaque mot de code, en ne gardant que les dictionnaires incomplets potentiellement préfixes.

Comme dans le cas de la déduction par mot de code décrite dans le paragraphe 5.3.1.1, la taille de l'espace de recherche peut être réduite en exploitant, le fait qu'inverser tous les bits d'un dictionnaires ne change pas ses propriétés de distance, et en considérant aussi l'ordre lexicographique des mots de code de même longueur.

TABLE 5.3 – Déduction de dictionnaire en spécifiant tous les bits d'un plan de bits

$\left\{\mathcal{C}^{(1)},\ldots,\mathcal{C}^{(n)}\right\}$ =déduction_par_plan_de_bits($\mathcal{C}$)

si $\mathcal{C}$ est totalement déterminé, $\mathcal{S}=\{\mathcal{C}\}$, renvoyer $\mathcal{S}$.
Déterminer k, l'indexe du premier plan de bit indéterminé dans $\mathcal{C}$;
Déterminer i, l'indexe du plus court mot de code non déterminé ;
$\mathcal{S}=\{\}$;
pour $j=0$ à 2^{M-i}
    $\mathcal{C}^{(j)}=\mathcal{C}$;
    pour $\lambda=i$ à M
        $c^{(j)}_{\lambda,k}=(j\&\,(1\ll(\lambda-i)))\gg(\lambda-i)$;
    fin pour
    Partitionner $\mathcal{C}^{(j)}$ en sous-dictionnaires
    en fonction de leur préfixe $\mathcal{C}^{(j)}_{1,k}\cup\cdots\cup\mathcal{C}^{(j)}_{\nu_j,k}$;
    Si tous les $\mathcal{C}^{(j)}_{m,k}$ satisfont (5.5), $\mathcal{S}=\mathcal{S}\cup\left\{\mathcal{C}^{(j)}\right\}$;
    fin Si
fin pour
renvoyer $\mathcal{S}$.

5.3.1.3 Construction en utilisant les arbres des codes canoniques

Comme nous l'avons rappelé dans le paragraphe 4.3.4, tout CLV préfixe conjoint peut être représenté par un arbre binaire étiqueté. Cependant, la structure de l'arbre de code non étiqueté donne déjà des informations concernant le code. Elle permet en particulier de déduire des bornes sur la distance libre des codes associés à cet arbre.

Pour exploiter efficacement ce fait, les arbres de code sont regroupés dans des classes d'équivalence. Une classe d'équivalence contient des arbres qui ne différent que par leur étiquetage. Ceci est un problème classique d'isomorphisme d'arbres [AHU74, Bus97]. Les classes sont représentées par des *arbres canoniques* et peuvent être arrangées dans un *arbre d'arbres canoniques* (à ne pas confondre avec l'arbre de code). Chaque classe peut ainsi être explorée en utilisant une variante des deux méthodes exposées dans les paragraphes 5.3.1.1 et 5.3.1.2.

L'équivalence entre arbres de code est définie de manière inductive. Deux arbres S et T sont équivalents ($S\equiv T$) s'ils sont composés d'un seul noeud, ou s'ils ont les

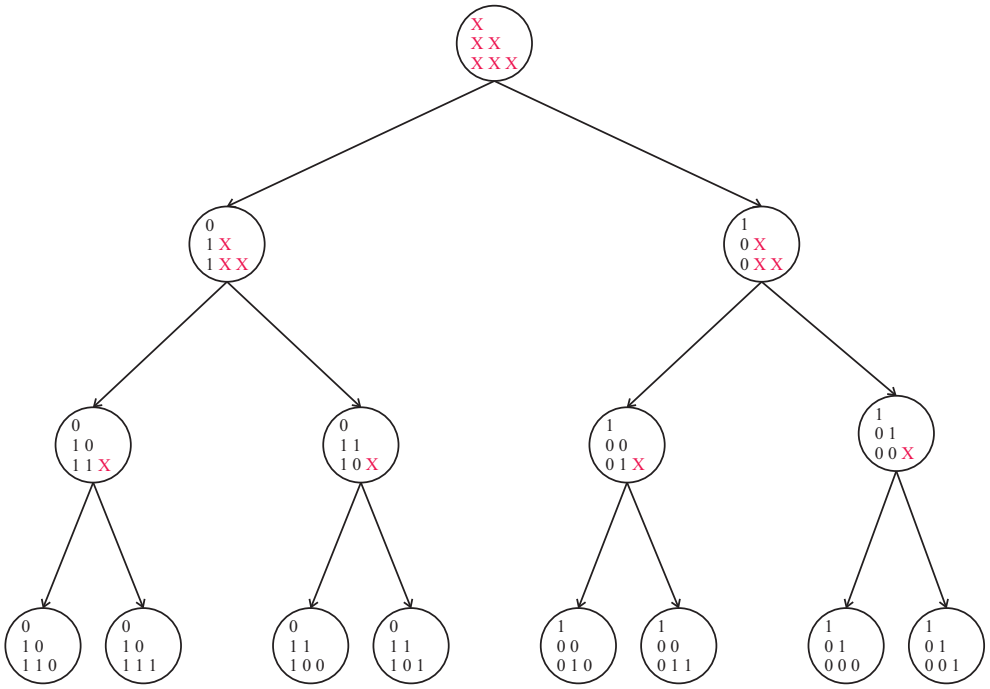


FIGURE 5.3 – Exemple d'un arbre de recherche de CLVs conjoints pour le vecteur de kraft $\boldsymbol{\ell} = (1, 2, 3)$ lorsque la déduction est faite par plan de bits

mêmes sous-arbres immédiats $S_1, S_2, \ldots, S_m$ (enraciné dans les fils direct de S) et $T_1, T_2, \ldots, T_m$, qui peut être ordonné de tel sorte que $S_i \equiv T_i$ pour tout $1 \leq i \leq m$ (définition adaptée de [Bus97]). En d'autres termes, deux arbres de code binaires sont équivalent s'il existe un isomorphisme qui transforme un dans l'autre en transposant les fils directs des noeuds intermédiaires y compris la racine. En supposant que l'arbre est construit de haut en bas à partir de la racine, tous les noeuds restent à leur niveaux, seules leur position horizontale change.

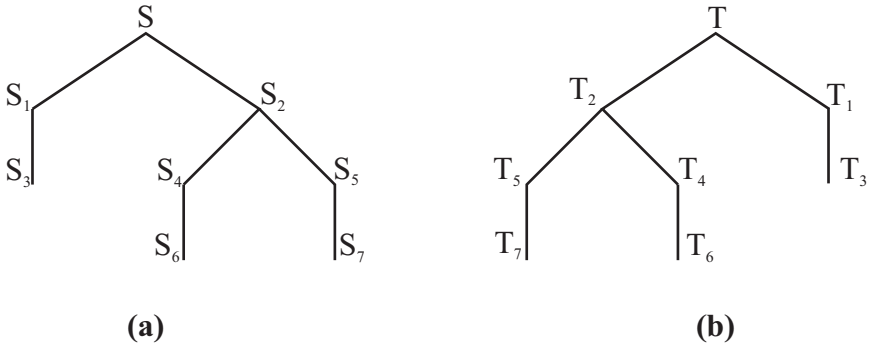


FIGURE 5.4 – Exemples d'arbres équivalents

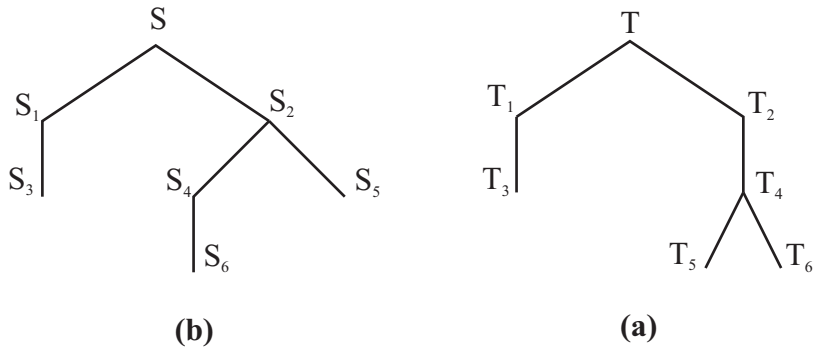


FIGURE 5.5 – Exemples d'arbres non équivalents

Les figures 5.4 (a) et 5.4 (b) montrent deux arbres T et S qui sont équivalents. Les figures 5.5 (a) et 5.5 (b) montrent deux arbres S et T non équivalents car on a $S_2 \not\equiv T_2$.

Pour un vecteur de Kraft donné $\boldsymbol{\ell}$, plusieurs arbres binaires correspondant à des codes préfixes peuvent être construits. Par exemple, la figure 5.6 fournit huit représentations pour $\boldsymbol{\ell} = (2, 3, 3)$. Les arbres (a) et (b) sont équivalents, tandis que (a) et (c) ne sont pas équivalents. Les arbres (c) et (d) sont aussi équivalents, les arbres (e) et (f) sont équivalents, et finalement les arbres (g) et (h) sont équivalents. Les dictionnaires correspondants aux mêmes arbres équivalents ont la même structure préfixe. Par exemple, les dictionnaires associés à la représentation (a) de la figure 5.6 sont de la forme $\mathcal{C}^{(a)} = \{c_1^1 c_1^2, c_1^1 \bar{c}_1^2 c_2^3, c_1^1 \bar{c}_1^2 \bar{c}_2^3\}$ alors que ceux de la représentation (e) sont de la forme $\mathcal{C}^{(e)} = \{c_1^1 c_1^2, \bar{c}_1^1 c_2^2 c_2^3, \bar{c}_1^1 \bar{c}_2^2 c_3^3\}$.

Comme spécifié ci-dessus, les arbres de code équivalents sont groupés dans une classe d'équivalence, et cette dernière peut être représentée par un *arbre canonique*. Le problème principale revient à énumérer les représentations de tous les arbres canoniques associés à un vecteur de Kraft donné $\boldsymbol{\ell}$. A notre connaissance, aucun algorithme n'est directement (et efficacement) applicable dans le cas où un vecteur de Kraft est donné.

Soit T un arbre binaire, left(T) son sous-arbre immédiat gauche et right(T) son sous-arbre immédiat droit (pour la compacité des notations, on utilise T pour nommer un arbre et sa racine). Soit la fonction $\boldsymbol{\ell}^*(T)$ qui donne le vecteur de Kraft associé à l'arbre T dans un ordre décroissant (le symbole * sera utilisé tout au long de ce paragraphe pour designer un vecteur dont les composants sont rangés dans un ordre décroissant). Par exemple, $\boldsymbol{\ell}^*(T) = (3, 3, 2, 1)$, T représente quatre mots de

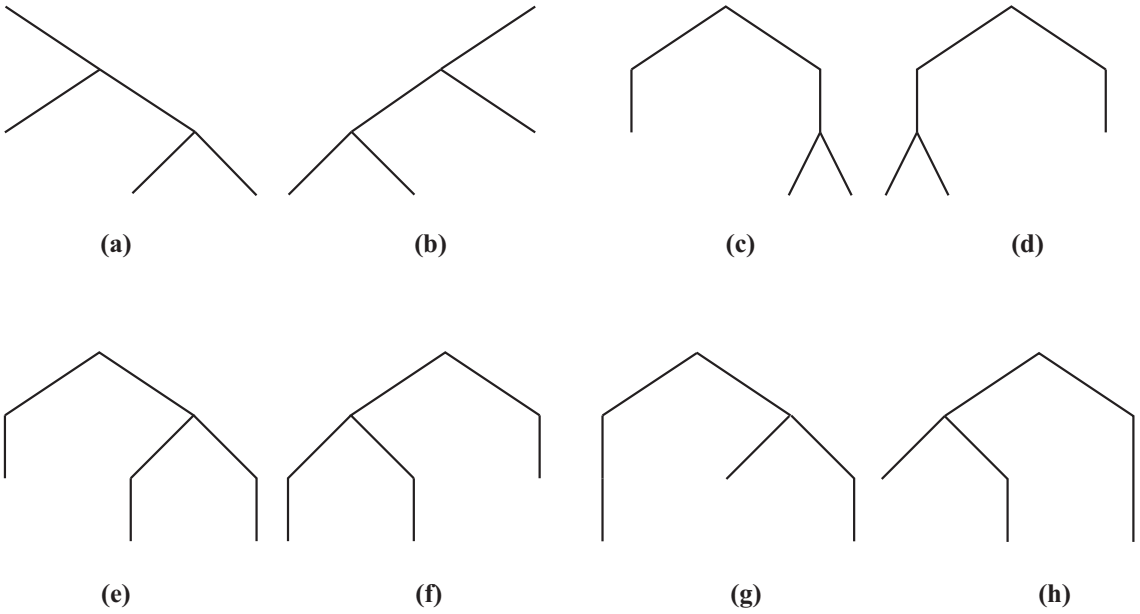


FIGURE 5.6 – Huit représentations d'arbre associés au vecteur de Kraft $\boldsymbol{\ell} = (2, 3, 3)$

code de longueur $1, 2, 3$ et 3. La somme de Kraft est définie comme

$$\text{Kraft}(\boldsymbol{\ell}^*(T)) = \sum_{i=1}^{M} 2^{-\ell_i^*}. \tag{5.6}$$

Par exemple, $\text{Kraft}((2, 1)) = 0.75$, $\text{Kraft}((0)) = 1$, où (0) représente un arbre composé seulement du noeud racine. Définissons l'ordre $\preceq$ entre arbres comme :

$$T_1 \preceq T_2 \quad \text{si et seulement si} \quad \boldsymbol{\ell}^*(T_1) \prec_{\text{lex}} \boldsymbol{\ell}^*(T_2), \tag{5.7}$$

où $\prec_{\text{lex}}$ est l'ordre lexicographique sur des vecteurs d'entiers. Par exemple, $(1) \prec_{\text{lex}} (1)$, $(3) \prec_{\text{lex}} (3, 2, 2)$, $(0) \prec_{\text{lex}} (1)$, et $() \prec_{\text{lex}} (0)$, où $()$ signifie pas d'arbre, c'est-à-dire un arbre vide qui est plus petit que n'importe quel arbre.

Definition 5.1 *Un arbre binaire est* canonique *s'il satisfait* $\text{left}(T_i) \preceq \text{right}(T_i)$ *à tous ses noeuds intermédiaires.*

Cette définition inductive garantit que l'arbre soit l'unique arbre canonique dans la classe d'équivalence, désigné par l'arbre minimal selon la relation d'ordre $\preceq$. Un arbre canonique peut être représenté à l'aide d'une liste en le parcourant dans n'importe quel ordre bien défini qui visite chaque noeud intermédiaire T_i une fois et en énumérant $(\boldsymbol{\ell}^*(\text{left}(T_i)), \boldsymbol{\ell}^*(\text{right}(T_i)))$. Une représentation à l'aide d'une chaine de

caractères compacte de l'arbre T utilisant les symboles '(','0',')' et ',' est obtenue en remplaçant récursivement $\boldsymbol{\ell}^*(T)$ par $(\boldsymbol{\ell}^*(\text{left}(T)), \boldsymbol{\ell}^*(\text{right}(T)))$.

Nous proposons la méthode suivante pour énumérer tous les arbres canoniques pour un vecteur de Kraft $\boldsymbol{\ell}^*$. L'énumération prendra la forme d'un *arbre de représentations canoniques* Σ de hauteur

$$L = \max_i \{\ell_i^*\} = \ell_1^*, \tag{5.8}$$

dont les feuilles correspondent aux arbres canoniques. Dans Σ, le passage d'un noeud père vers ses noeuds fils inclus un ensemble contenant les subdivisions des vecteurs de Krafts en deux parties (les sous-arbres) ordonnées par $\preceq$. En commençant avec le vecteur de Kraft $\boldsymbol{\ell}^*$ à la racine de Σ, on subdivise $\boldsymbol{\ell}^* - \mathbf{1}$ (soustraire 1 à toutes les longueurs pour descendre d'un niveau dans l'arbre) en deux parties $\boldsymbol{\ell}_1^*$ et $\boldsymbol{\ell}_2^*$, de telle sorte que $\boldsymbol{\ell}_1^* \prec_{\text{lex}} \boldsymbol{\ell}_2^*$ et $K(\boldsymbol{\ell}_i^*) \leqslant 1$ $(i = 1, 2)$. Étant donné que $\boldsymbol{\ell}_1^*$ peut être vide (correspondant à une feuille absente, c'est-à-dire une entrée -1 dans $\boldsymbol{\ell}^* - \mathbf{1}$), on a besoin de définir $K(()) = 1$. Le premier niveau en dessous de Σ contient toutes les subdivisions de $\boldsymbol{\ell}^*$, le second niveau contient des paires de subdivisions de $(\boldsymbol{\ell}_1^*, \boldsymbol{\ell}_2^*)$, et ainsi de suite ; le processus de subdivisions récursives s'arrête avec les feuilles '(0)' et les noeuds vides '()'.

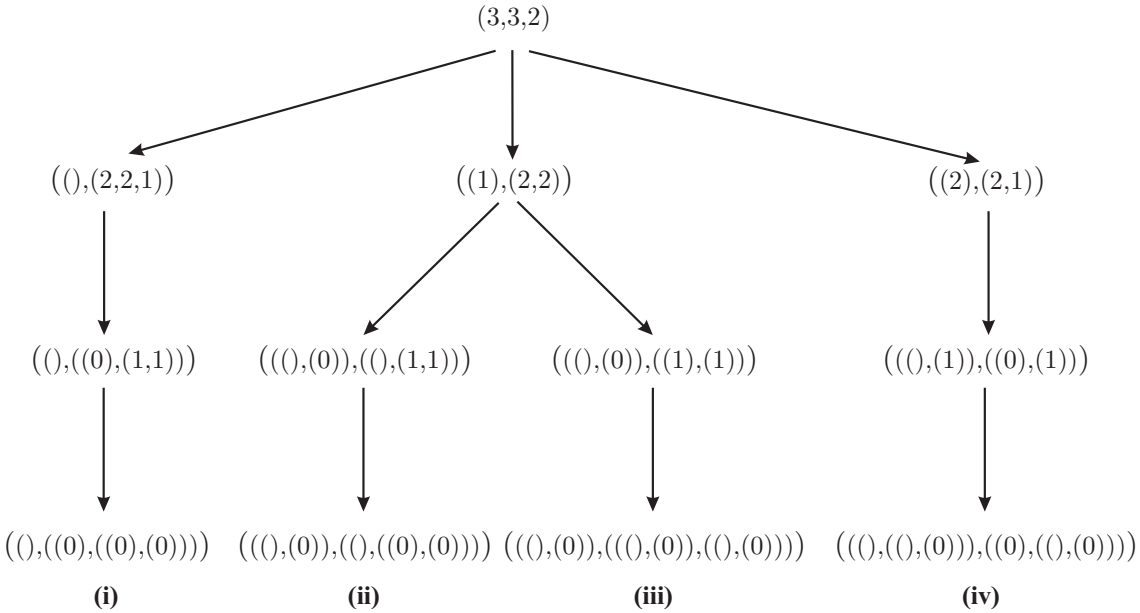


FIGURE 5.7 – Arbre des représentations canoniques de $\boldsymbol{\ell}^* = (3, 3, 2)$

Exemple 5.5 *Soit* $\boldsymbol{\ell} = (2, 3, 3)$ *un vecteur de Kraft. La figure 5.7 montre l'arbre des représentations canoniques pour* $\boldsymbol{\ell}^* = (3, 3, 2)$. *En partant du noeud racine* $\boldsymbol{\ell}^*$, *on a* $\boldsymbol{\ell}^* - 1 = (2, 2, 1)$. *Il y a trois choix pour* $(\boldsymbol{\ell}_1^*,\ \boldsymbol{\ell}_2^*)$ *au premier niveau :* $(\boldsymbol{\ell}_1^* = (),\ \boldsymbol{\ell}_2^* = (2, 2, 1))$, $(\boldsymbol{\ell}_1^* = (1),\ \boldsymbol{\ell}_2^* = (2, 2))$ *et* $(\boldsymbol{\ell}_1^* = (2),\ \boldsymbol{\ell}_2^* = (2, 1))$. *Chaque choix conduit à un ou plusieurs arbres canoniques. Au second niveau, il y a par exemple, deux manière de diviser la paire* $((1),\ (2, 2))$. *Ce processus est répété pour tous les noeuds intermédiaires. Ce qui conduit à la première branche à gauche notée (i) sur la figure 5.7 qui est la représentation de l'arbre canonique de la figure 5.6 (a). Les branches (ii), (iii) et (iv) de la figures 5.7 correspondent aux représentations des arbres canoniques respectifs des arbres binaires des figures 5.6 (c), (e) et (g.)*

Comme illustré par la figure 5.7, les noeuds intermédiaires de l'arbre des représentations canoniques correspondent à des *représentations canoniques partiellement déterminées*. Par exemple, la représentation $((1), (2, 2))$ dans la branche du milieu de la figure 5.7 indique que le deuxième et le troisième mot de code ont un préfixe commun, conduisant à des dictionnaires de la forme $\mathcal{C}^1 = \{c_1^1 c_1^2, \bar{c}_1^1 c_2^2 c_2^3, \bar{c}_1^1 c_3^2 c_3^3\}$, alors que la représentation $(((), (0)), ((), ((0), (0))))$ conduit à des dictionnaires de la forme $\mathcal{C}^2 = \{c_1^1 c_1^2, \bar{c}_1^1 c_2^2 c_2^3, \bar{c}_1^1 c_2^2 \bar{c}_2^3\}$.

En étendant la notion de déduction de dictionnaire (définition 4.2 à la page 68) à des étiquettes symboliques, on constate que $\mathcal{C}^1 \prec \mathcal{C}^2$. Donc les bornes de la distance libre calculées pour $\mathcal{C}^1$ resteront valides aussi pour $\mathcal{C}^2$ et pour tous les codes qui en dérivent. En effet il y a une correspondance entre la représentation canonique partiellement déterminée T^i et le dictionnaire associé $\mathcal{C}^i$ avec des étiquettes symboliques.

Ainsi, si l'arbre T^1 est le fils de l'arbre T^0 dans Σ, alors d'après le corollaire 4.4 (page 78) les bornes sur la distance libre définies dans le paragraphe 4.3.4 satisfont

$$[\underline{d}^{\mathrm{T}}_{\mathrm{libre}}(T^1), \overline{d}^{\mathrm{T}}_{\mathrm{libre}}(T^1)] \subseteq [\underline{d}^{\mathrm{T}}_{\mathrm{libre}}(T^0), \overline{d}^{\mathrm{T}}_{\mathrm{libre}}(T^0)]. \tag{5.9}$$

Pour un vecteur de Kraft donné $\boldsymbol{\ell}$, nous parcourons l'arbre des représentations canoniques (figure 5.7) avec l'algorithme du *branch-and-prune*. Lorsque un arbre canonique entièrement déterminé (une feuille) est atteint, toutes les étiquettes possibles de cet arbre canonique sont explorées dans un mode *branch-and-prune* en utilisant une des méthodes décrites dans les paragraphes 5.3.1.1 et 5.3.1.2.

5.3.2 Arbre de codes arithmétiques

On détaille maintenant la structuration en arbre des automates implémentant des codeurs arithmétiques. Par soucis de clarté, la méthode de synthèse est décrite pour des sources binaires ($M = 2$). La généralisation à plusieurs symboles est possible.

Considérons une source X avec l'alphabet $\mathcal{A} = \{a_0, a_1\}$ auquel on associe l'ensemble de probabilités $\mathcal{P} = \{p_0, p_1\}$. On a $\Pr(X = a_0) = p_0$ et $\Pr(X = a_1) = p_1$. Soit (l, h, f) l'état courant du codeur et $w = h - l$ la largeur de l'intervalle de codage courant lors du processus de CA en précision finie (voir les paragraphes 2.2.2.3 et 2.2.2.5). Durant le codage, $[l_i, h_i)$ est le sous-intervalle de largeur $w_i = h_i - l_i$ assigné au symbole a_i pour $i \in \{0, 1\}$.

Un CA conjoint peut être obtenu à partir d'un codeur arithmétique en précision finie en considérant un symbole interdit (SI), voir les paragraphes 2.2.2.4 et 2.2.2.5. Dans le cas d'un seul SI, soit p_ε la "probabilité" de ce SI et w_ε la largeur du sous-intervalle qui lui est assigné. Pour une valeur donnée de p_ε et p_0, les largeurs des sous-intervalles de l'intervalle de codage $[l, h)$ sont calculées de la manière suivante

$$w_\varepsilon = \langle p_\varepsilon \times w \rangle \,, \tag{5.10}$$

$$w_0 = \langle p_0 \times (h - l - w_\varepsilon) \rangle \,, \tag{5.11}$$

$$w_1 = h - l - w_0 - w_\varepsilon, \tag{5.12}$$

où $\langle \cdot \rangle$ signifie arrondir vers l'entier le plus proche.

Pour augmenter les performances de correction d'erreurs d'un CA conjoint en précision finie sans augmenter la probabilité allouée au SI, la probabilité du SI peut être distribuée à $M + 1 = 3$ SIs, $\{\varepsilon_0, \varepsilon_1, \varepsilon_2\}$, avec les probabilités correspondantes $\{p_{\varepsilon_0}, p_{\varepsilon_1}, p_{\varepsilon_2}\}$ de telle sorte que $p_{\varepsilon_0} + p_{\varepsilon_1} + p_{\varepsilon_2} = p_\varepsilon$.

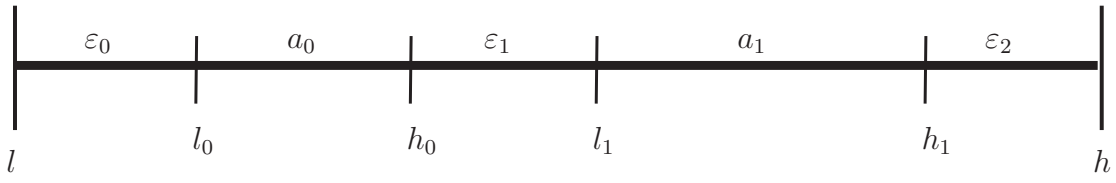


FIGURE 5.8 – Subdivision de l'intervalle courant de codage dans le cas le plus général du CA conjoint en précision finie à entrée binaire et trois symboles interdits.

La figure 5.8 montre comment l'intervalle de codage peut être subdivisée durant le processus de codage dans le cas d'un CA conjoint avec trois symboles interdits.

La manière de distribuer p_ε entre $\{\varepsilon_0, \varepsilon_1, \varepsilon_2\}$ peut être indépendante de l'état du codeur, c'est-à-dire indépendante des valeurs de l, h et f, ou peut avoir une *dépendance vis-à-vis de l'état*. Dans ce dernier cas, l'ordre des sous-intervalles assignés aux symboles source peut changer d'un état à un autre. La dépendance vis-à-vis de l'état dans l'assignation de la probabilité du symbole interdit fournit encore plus de degrés de liberté pour construire des codes potentiellement meilleurs que ceux obtenus avec une indépendance vis-à-vis de l'état qui sont déjà considérés dans [BJWK08].

A notre connaissance, aucun outil analytique n'est disponible pour trouver la meilleure assignation de probabilité entre $\{\varepsilon_0, \varepsilon_1, \varepsilon_2\}$ si l'on considère que cette assignation peut être une fonction de l'état.

En commençant par l'automate incomplet avec l'état $(0, T, 0)$, l'ensemble de tous les codeurs peut être obtenu en explorant itérativement les successeurs de tous les états terminaux.

5.3.2.1 Exploration d'un état terminal

Chaque état terminal sera exploré avec toutes les configurations possibles des sous-intervalles $[l_0, h_0)$ (associé a_0) et $[l_1, h_1)$ (associé à a_1) de $[l, h)$ tel que $[l_0, h_0)$ et $[l_1, h_1)$ ne se chevauchent pas (de manière à garder les CAs décodables de manière instantanée). Ceci peut être fait en permettant à l_0 et l_1 de varier de l à $h-1$ par pas de 1 et en ne gardant que les configurations pour lesquelles $[l_0, h_0)$ et $[l_1, h_1)$ ne se chevauchent pas, c'est-à-dire, celles qui satisfont une des conditions suivantes,

$$l \leqslant l_0 < (h_0 = l_0 + w_0) \leqslant l_1 < (h_1 = l_1 + w_1) \leqslant h, \tag{5.13}$$

$$l \leqslant l_1 < (h_1 = l_1 + w_1) \leqslant l_0 < (h_0 = l_0 + w_0) \leqslant h. \tag{5.14}$$

Exemple 5.6 *Soit le code $\mathcal{C}^0$ généré par un CA en précision finie dont les paramètres du codeur sont les suivants : $\left\{T = 8, p_0 = \frac{1}{4}, p_1 = \frac{3}{4}, p_\varepsilon = \frac{1}{2}, f_{\max} = 2\right\}$. La figure 5.9 montre quelques subdivisions possibles pour l'état initial de codage $(0, T, 0)$ menant à des automates incomplets totalement différents.*

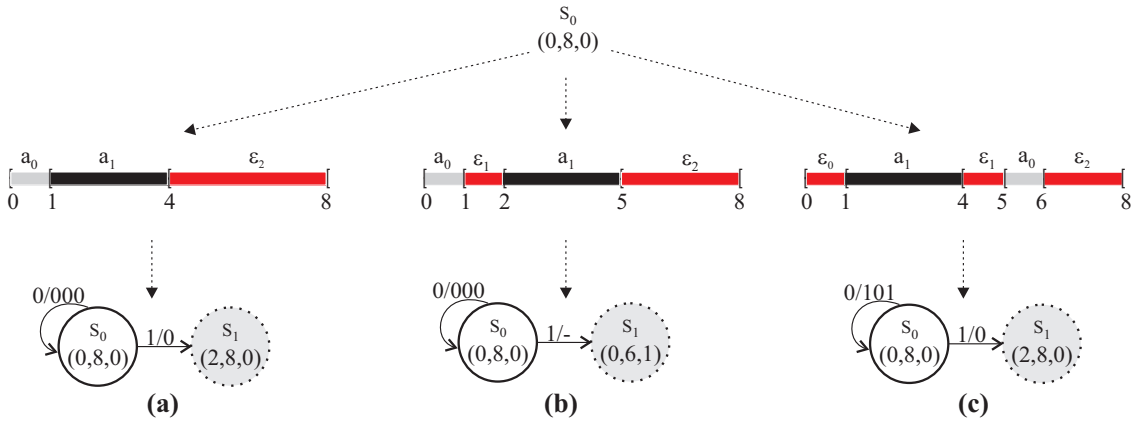


FIGURE 5.9 – Quelques subdivisons possibles de l'intervalle initial de codage pour le CA conjoint en précision finie dont les paramètres du codeur sont $T = 8$, $p_0 = \frac{1}{4}$, $p_1 = \frac{3}{4}$, $p_\varepsilon = \frac{1}{2}$ et $f_{\max} = 2$

- *Dans la première subdivision (a) à gauche, le sous-intervalle* $[l_0, h_0) = [0, 1)$ *a été assigné au symbole* a_0 *et le sous-intervalle* $[l_1, h_1) = [1, 4)$ *a été assigné au symbole* a_1*, c'est-à-dire* $p_{\varepsilon_0} = 0$*,* $p_{\varepsilon_1} = 0$ *et* $p_{\varepsilon_2} = p_\varepsilon$*.*
- *Dans la subdivision du milieu (b),* $[l_0, h_0) = [0, 1)$ *et* $[l_1, h_1) = [2, 5)$*.*
- *Dans la troisième subdivision* $[l_0, h_0) = [5, 6)$ *et* $[l_1, h_1) = [1, 4)$*.*

Les largeurs des sous-intervalles $[l_0, h_0)$ *et* $[l_1, h_1)$ *restent inchangées dans toutes les configurations, seules, leurs bornes changent.*

5.3.2.2 Construction de l'arbre des automates

A chaque fois que (5.13) ou (5.14) sont satisfaites, un nouvel automate complet ou incomplet est obtenu à partir de l'automate incomplet précédant en le complétant de deux états (obtenus à partir de $[l_0, h_0)$ et $[l_1, h_1)$ après les normalisations appropriées) s'ils n'existent pas déjà. Les automates incomplets résultants sont explorés à leur tour. A partir de l'automate incomplet initial avec l'état $(l = 0, h = T, f = 0)$, l'expansion des automates incomplets donne un arbre d'automates dans lequel les noeuds intermédiaires correspondent à des automates incomplets et les feuilles correspondent à des automates complets et chaque branche correspond à une exploration possible d'un état terminal.

La figure 5.10 montre comment tous les automates pour des valeurs des pa-

ramètres caractéristiques fixées avec une dépendance vis-à-vis de l'état du codeur peuvent être décrits par un arbre. L'automate incomplet initial composé de l'état non exploré $(0, T, 0)$ est hérité par tous les automates. Chaque noeud dans la première couche des noeuds intermédiaires représente un automate incomplet pour lequel une configuration donnée des sous-intervalles de l'intervalle de code initial a été considérée. La figure 5.11 montre un exemple d'un arbre d'automates de l'exemple 5.6. Les

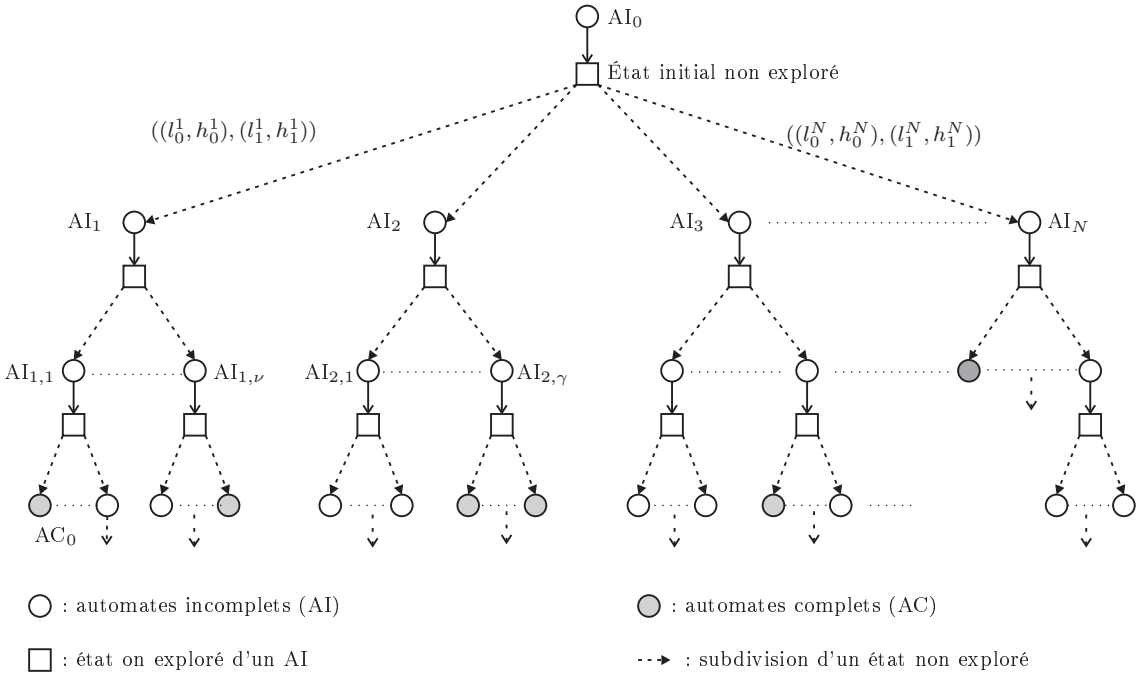


FIGURE 5.10 – Hiérarchisation de tous les CEFs dans un arbre pour des valeurs des paramètres caractéristiques fixées

cercles étiquetés $s_0, s_1, \ldots$ représentent les états des automates (in)complets. Ils sont colorés en gris pour les états non explorés. L'automate incomplet initial ne contient que l'état initial $s_0 = (0, 8, 0)$.

Une possibilité d'étendre l'état initial est d'assigner à a_0 l'intervalle $[0, 1)$ et à a_1 l'intervalle $[1, 4)$ (voir figure 5.9 (a)) conduisant au second automate incomplet AI$_1$. La dernière extension possible de l'état initial associe à a_0 l'intervalle $[7, 8)$ et à a_1 l'intervalle $[4, 7)$ produisant l'automate incomplet AI$_N$. En répétant cette approche de manière itérative, on peut trouver les automates complets comme celui en gris dans la partie inférieure gauche de l'arbre.

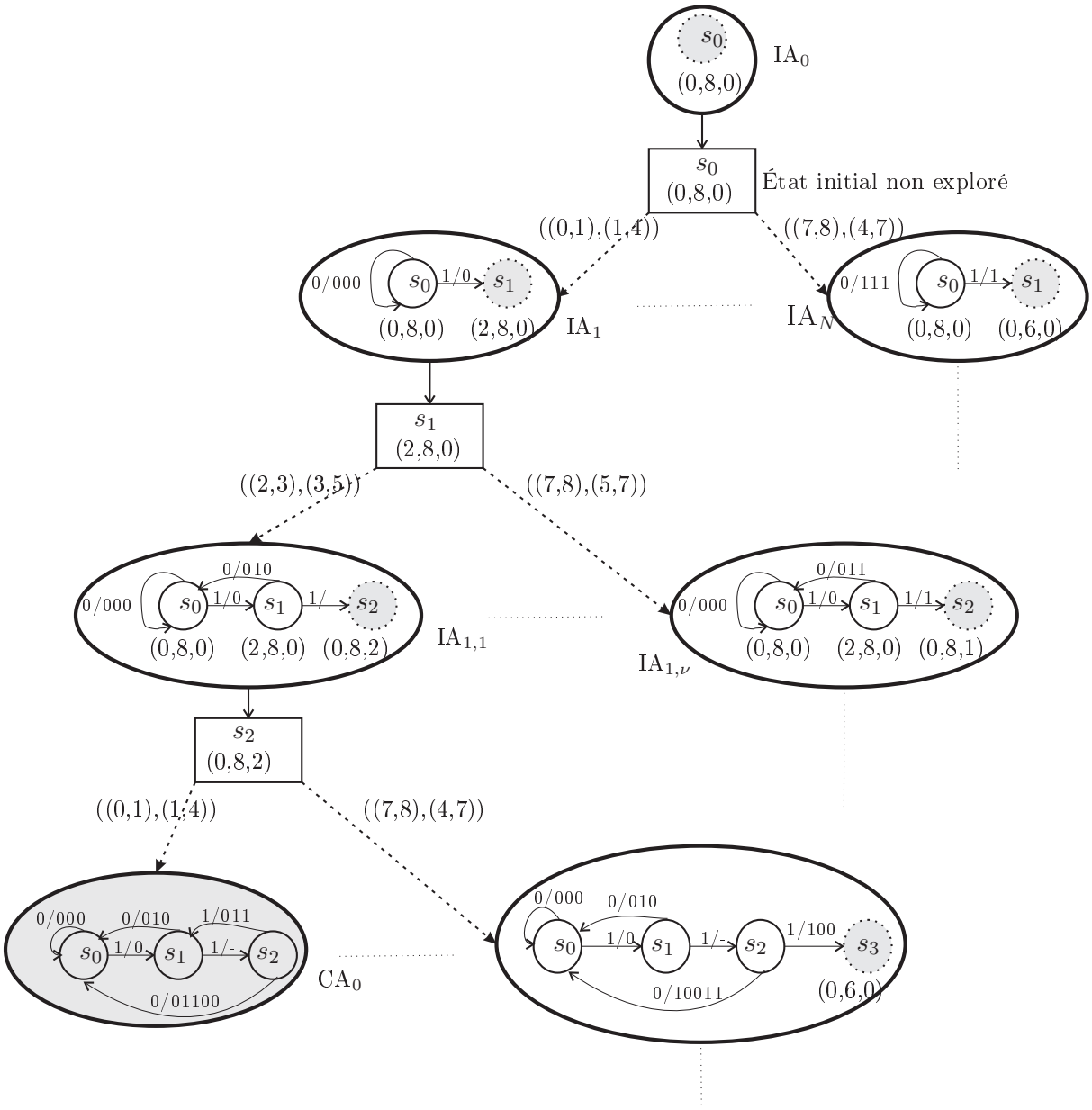


FIGURE 5.11 – Une partie de l'arbre des automates pour l'exemple 5.6 ; les conventions de la figure 5.10 sont utilisées, les étiquettes sur les flèches en pointillés représentent les intervalles alloués $((l_0, h_0), (l_1, h_1))$ aux symboles a_0 et a_1.

5.3.2.3 Extension aux CAs conjoints à entrée non binaire

Pour les sources avec $M > 2$, étendre la subdivision de l'intervalle présentée au paragraphe 5.3.2, exigerait de considérer $M + 1$ symboles interdits. Autoriser une dépendance vis-à-vis de l'état pour l'assignation des probabilités des symboles interdits, peut conduire à un très grand nombre d'automates pour des valeurs des paramètres caractéristiques données. Cependant, dans la pratique, la plupart (voire tous) des codeurs de sources standardisés incluant un codage arithmétique, introduisent une étape de binarisation sur les symboles non binaires à coder avant un codage arithmétique à entrées binaires [Ric03]. Ceci permet l'utilisation d'un codeur quasi-arithmétique avec une précision réduite. A la place d'un seul modèle de probabilité de la source, plusieurs modèles adaptatifs de probabilités sont considérés et choisis en fonction d'un *contexte*, correspondant ici à l'index du bit à coder dans la d'un symbole binarisé. Les méthodes décrites dans le paragraphe 5.3.2 sont ainsi étendues pour des sources binarisées avec $M > 2$ symboles en introduisant un contexte simple, sans adaptativité des modèles de probabilités.

Soit X une source sans mémoire à valeurs dans $\mathcal{A}_M = \{a_1, a_2, \ldots, a_M\}$ et $\boldsymbol{P}_a = (p_{a_1}, \ldots, p_{a_M})$ l'ensemble des probabilités des symboles correspondant. Sans perte de généralité, supposons que $p_{a_i} \geq p_{a_{i+1}}$, $i = 1, \ldots, M$.

Une binarisation de la source $\mathcal{A}_M$ peut être faite de la manière suivante. Considérons $\mathrm{B}_L(x)$ (voir le paragraphe 5.3.1.1 à la page 89) qui est la représentation binaire de l'entier $x \in \mathbb{N}$ sur L bits, avec $L \in \mathbb{N}^*$ tel que $L \geq \lceil \log_2(x) \rceil$. Par exemple $\mathrm{B}_3(1) = 001$. Maintenant, considérons $L_M \in \mathbb{N}^*$ tel que $L_M \geq \lceil \log_2(M) \rceil$. Pour chaque symbole $a_i \in \mathcal{A}_M$, $\mathrm{B}_{L_M}(i-1)$ est une représentation binaire possible.

Exemple 5.7 *Considérons les 26 symboles de l'alphabet anglais $\mathcal{A}_{26}$. Alors $M = 26$ et $L_M = 5$ bits. Le tableau 5.4 montre les probabilités d'occurrence et l'assignation binaire de chaque symbole dans $\mathcal{A}_{26}$. L'entropie d'une source X sans mémoire générant les symboles en fonction des probabilités dans le tableau 5.4 est $H(X) = 4.175$ bits/symbole.*

Soit a_i^j, $i = 1, \ldots, M$ et $j = 1, \ldots, L_M$ le $j^{\text{ième}}$ bit de la représentation binaire du symbole a_i. Pour éviter la confusion avec les sorties binaires de codeur arithmétique, les a_i^j sont appelés *symboles*.

Soit p_b^j pour $b \in \{0, 1\}$ et $j = 1, \ldots, L_M$ la probabilité que $a_i^j = b$. On a $p_b^j =$

TABLE 5.4 – Probabilité d'occurrence de chaque symbole de l'alphabet prise dans [But95] et un exemple de binarisation

Symboles	Probabilités	Assignation de bits
$a_1 = E$	$p_{a_1} = 0.1270$	0 0 0 0 0
$a_2 = T$	$p_{a_2} = 0.0906$	0 0 0 0 1
$a_3 = A$	$p_{a_3} = 0.0817$	0 0 0 1 0
$a_4 = O$	$p_{a_4} = 0.0751$	0 0 0 1 1
$a_5 = I$	$p_{a_5} = 0.0697$	0 0 1 0 0
$a_6 = N$	$p_{a_6} = 0.0674$	0 0 1 0 1
$a_7 = S$	$p_{a_7} = 0.0633$	0 0 1 1 0
$a_8 = H$	$p_{a_8} = 0.0609$	0 0 1 1 1
$a_9 = R$	$p_{a_9} = 0.0599$	0 1 0 0 0
$a_{10} = D$	$p_{a_{10}} = 0.0425$	0 1 0 0 1
$a_{11} = L$	$p_{a_{11}} = 0.0403$	0 1 0 1 0
$a_{12} = C$	$p_{a_{12}} = 0.0278$	0 1 0 1 1
$a_{13} = U$	$p_{a_{13}} = 0.0276$	0 1 1 0 0
$a_{14} = M$	$p_{a_{14}} = 0.0241$	0 1 1 0 1
$a_{15} = W$	$p_{a_{15}} = 0.0236$	0 1 1 1 0
$a_{16} = F$	$p_{a_{16}} = 0.0223$	0 1 1 1 1
$a_{17} = G$	$p_{a_{17}} = 0.0202$	1 0 0 0 0
$a_{18} = Y$	$p_{a_{18}} = 0.0197$	1 0 0 0 1
$a_{19} = P$	$p_{a_{19}} = 0.0193$	1 0 0 1 0
$a_{20} = B$	$p_{a_{20}} = 0.0149$	1 0 0 1 1
$a_{21} = V$	$p_{a_{21}} = 0.0098$	1 0 1 0 0
$a_{22} = K$	$p_{a_{22}} = 0.0077$	1 0 1 0 1
$a_{23} = J$	$p_{a_{23}} = 0.0015$	1 0 1 1 0
$a_{24} = X$	$p_{a_{24}} = 0.0015$	1 0 1 1 1
$a_{25} = Q$	$p_{a_{25}} = 0.001$	1 1 0 0 0
$a_{26} = Z$	$p_{a_{26}} = 0.0007$	1 1 0 0 1

$\sum_{i=1}^{M} p_{a_i}\delta(a_i^j - b)$, où $\delta(x)$ est une fonction indicatrice ($\delta(x) = 1$ si $x = 0$ et $\delta(x) = 0$ sinon). On a bien sûr $p_0^j + p_1^j = 1$.

L'entropie d'une source binaire sans mémoire X_2^j générant le $j^{\text{ième}}$ bit d'un symbole binarisé avec le modèle de probabilités $\left\{p_0^j, p_1^j\right\}$ peut être facilement évaluée par :

$$H(X_2^j) = -p_0^j \log_2(p_0^j) - p_1^j \log_2(p_1^j). \tag{5.15}$$

L'entropie de la source après binarisation est ainsi de $\sum_{j=1}^{L(M)} H(X_2^j)$.

Exemple 5.8 *Pour $\mathcal{A}_{26}$, $p_0^1 = 0.0963$; $p_0^2 = 0.269$; $p_0^3 = 0.379$; $p_0^4 = 0.432$; $p_0^5 = 0.455$ et $\sum_1^5 H(X_2^j) = 4.237$ bits/symbole non binarisé, qui est supérieure à $H(X)$. Utiliser un codeur arithmétique incluant cinq modèles de probabilité indépendants pour des sources sans mémoire conduit ainsi à une perte d'efficacité de compression comparée à celle d'un codeur arithmétique appliqué directement aux M symboles de la source.*

Avec un contexte qui correspond à l'indexe du bit à coder dans le symbole source binarisé, le symbole a_i^j est codé avec le modèle de probabilité $\left\{p_0^j, p_1^j\right\}$. Pour le codeur à états finis, cela exige de garder en mémoire le contexte par exemple en l'ajoutant à l'état $\{l, h, f\}$ pour obtenir le nouvel état $\{l, h, f, j\}$, avec $j = 1, \ldots, L_M$. Cela a pour conséquence une multiplication par L_M du nombre d'états dans l'automate.

Lorsqu'on construit un CA conjoint, on essaye d'atteindre un certain taux de codage source-canal R_c. Pour cette raison, une probabilité

$$p_\epsilon^j = 1 - 2^{-(R_c - H(X_2^j))} \tag{5.16}$$

est assignée à chaque contexte j. Les paramètres caractéristiques d'un CA conjoint sont alors T, $f_{\max}$, $\boldsymbol{p}$, et R_c. Comme dans le cas d'une source binaire, l'ensemble de tous les codeurs qui peuvent être obtenus par une assignation des probabilités des symboles interdits avec une dépendance d'état peuvent être générés en explorant itérativement les successeurs de tous les états non explorés des automates incomplets, en commençant par l'automate incomplet initial ayant le seul état ($l = 0, h = T, f = 0, j = 1$), pour toutes les configurations admissibles des sous-intervalles d'un état non exploré.

5.4 Conclusion

Dans ce chapitre, nous avons décrit plusieurs techniques d'optimisation des codes conjoints. notre approche consiste à ordonner l'ensembles des codeurs à états finis dans une structure d'arbre à codeurs à états finis pour y appliquer un algorithme de type *branch-and-prune*. Trois méthodes de parcours de l'arbre à codeurs à états finis sont proposés : exploration en profondeur, exploration en largeur et exploration avec la méthode par tri.

Pour les CLVs conjoints, nous avons décrit trois méthodes permettant de les structurer sur un arbre : la *construction par mot de code*, la *construction par plan bits* et la *construction par arbre canonique*.

Pour les CAs conjoints, nous avons présenté une méthode qui consiste à utiliser la notion de symbole interdit multiples et une dépendance vis-à-vis de l'état du codeur pour structurer l'arbre des automates à explorer.

Les performances des méthodes proposées seront évaluées dans le chapitre suivant.

Chapitre 6

Résultats expérimentaux

Ce chapitre présente des ensembles d'expériences qui permettent d'évaluer à la fois les performances de la méthode d'évaluation de distance présentée au chapitre 3 et les techniques d'optimisation de codes conjoints décrites au chapitre 5.

6.1 Codes arithmétiques

Deux ensembles d'expérimentations ont été menés. D'abord, des sources binaires simples sont considérées, permettant une évaluation simple de l'algorithme *branch-and-prune* comparé à une recherche exhaustive du meilleur code arithmétique (CA) conjoint. L'évaluation de la distance libre exploitant l'algorithme de Dijkstra (paragraphe 3.1, page 57) est comparé à la méthode décrite dans [BJWK08]. Les différentes méthodes de parcours d'arbre sont comparées. Ensuite, un CA conjoint pour l'alphabet anglais binarisé est fourni.

6.1.1 CA conjoint pour des sources binaires

Un premier CA conjoint avec les paramètres caractéristiques $T = 8$, $f_{\max} = 1$, $p_0 = 0.1$, et un $R_c = 0.62$ bits/symbole ($p_\varepsilon = 0.1$) est d'abord considéré. Le temps nécessaire pour générer tous les automates avec une dépendance vis-à-vis de l'état lors des assignations des probabilités des symboles interdits, calculer leurs distances libres et choisir le meilleur est de 6300 s. En utilisant l'algorithme *branch-and-prune* avec l'exploration en largeur, le temps nécessaire pour trouver la plus grande distance libre est seulement de 25 s, le temps de calcul est divisé par 252. Dans les deux cas, la

TABLE 6.1 – Comparaison entre les trois méthodes d'exploration de l'arbre pour $T = 16$, $\mathrm{P}_0 = 0.1$, $P_\varepsilon = 0.26$

Méthodes	exploration en profondeur	exploration en largeur	exploration selon la méthode par tri
$\lvert\mathcal{S}_r\rvert$	8	2	3
$\lvert\mathcal{T}_r\rvert$	28	9	12
d_{free}	3	3	3
R_{c}	0.93	0.92	0.92
Temps avec [BJWK08]	451266 s	31338 s	12431 s
Temps avec GDP	734 s	219 s	45 s

distance libre a été évaluée avec l'algorithme décrite dans [BJWK08].Cette première expérience a été faite avec un Intel Core 2 Duo à 2.66 Ghz avec 1 GO de mémoire.

Un second CA conjoint à entrées binaires avec les paramètres caractéristiques $T = 16$, $f_{\max} = 1$, $p_0 = 0.1$, et un taux de codage désiré $R_{\text{c}} = 0.91$ bits/symboles ($p_\varepsilon = 0.27$) est maintenant considéré. Un recherche exhaustive pour un tel CA conjoint prend un temps très important.

Le tableau 6.1 montre le temps nécessaire pour une exploration en profondeur, en largeur et la méthode par tri, décrites dans le paragraphe 5.1 pour trouver le meilleur automate. La méthode d'évaluation de la distance libre présentée dans [BJWK08] est comparée à celle présentée dans le paragraphe 3.1.

$|\mathcal{S}_r|$ et $|\mathcal{T}_r|$ désignent le nombre d'états et de transitions dans le codeur à états finis réduit. Ces nombres dépendent de la méthode d'exploration, puisque plusieurs codes à états finis peuvent avoir la même distance libre, sans nécessairement avoir le même nombre d'état et le même nombre de transitions.

Le taux de codage est exprimé en bits/symbole. La méthode par tri est la meilleure en ce qui concerne le temps d'exécution. Ceci est principalement du au fait que cette méthode explore en premier les automates incomplets qui ont le plus grand potentiel de conduire à la plus grande borne de la distance libre de telle sorte que $\underline{d}_{\text{libre}}$ peut être augmentée assez rapidement. Avoir une grande valeur de $\underline{d}_{\text{libre}}$ au début de l'algorithme, permet d'élaguer plus facilement de grandes partie de l'arbre de recherche de code.

On peut aussi constater qu'utiliser l'algorithme de Dijkstra sur le graphe des distances entre paires (GDP) pour calculer la distance libre est beaucoup plus efficace qu'utiliser la méthode de [BJWK08]. En effet Dijkstra sur le GDP est 614 fois plus rapide avec l'exploration en profondeur, 143 fois plus rapide avec l'exploration en

largeur et 276 fois plus rapide avec l'exploration avec la méthode par tri.

Lorsqu'on fait une comparaison du CA conjoint et du schéma tandem équivalent (CA en précision finie suivi par un code convolutif) avec le même taux de codage, la distance libre du CA conjoint obtenu reste inférieure. Pour l'exemple du tableau 6.1, considérons un schéma tandem de taux $R_c = 0.92$ bits/symboles équivalent, avec un CA en précision finie avec $T = 16$, $p_0 = 0.1$, $p_\varepsilon = 0$, suivi par un code convolutif de rapport $1/2$. La distance libre du schéma tandem dépend de la longueur de contrainte du code convolutif. Pour une longueur de contrainte de 2 (respectivement 3), la meilleure distance libre de code convolutif de rapport $1/2$ est de 4 (respectivement 5) [Pro95, chapitre 8]. La faiblesse du CA conjoint est principalement due à sa faible mémoire (qui est liée à ses états), qui est plus gérée pour une bonne compression que pour une grande distance libre.

Le nombre minimum d'états obtenus dans le CEF du CA conjoint est de 2, (voir le tableau 6.1), alors que dans le schéma tandem, le nombre total d'états est le produit du nombre d'états du CA en précision finie et le nombre d'états du code convolutif (au minimum 2). Par conséquent, le schéma conjoint sera moins complexe que le schéma tandem.

Dans le tableau 6.1, on note une légère variation du taux de compression qui est due aux effets d'arrondis du codage arithmétique en précision finie. Cette seconde expérience a été faite sur un Intel Xeon E5420 à 2.50GHz avec 64 GO de mémoire.

6.1.2 CA conjoint pour les sources non binaires

Maintenant, notre objectif est de construire de bons CA conjoint pour l'alphabet anglais binarisé $\mathcal{A}_{26}$ donné dans le tableau 5.4. Pour réduire le nombre d'automates (in)complets à construire dans l'arbre d'automates, seules deux valeurs de contextes sont considérées. Le premier contexte correspond à l'indice du premier bit et le second aux indices restants. Deux modèles de probabilités sont alors considérés nommés $\{p_0^1, p_1^1\}$ et $\{p_0^{2:5}, p_1^{2:5}\}$ avec

$$p_0^{2:5} = \frac{\sum_{j=2}^{5} p_0^j}{4} \quad \text{et} \quad p_1^{2:5} = 1 - p_0^{2:5}. \tag{6.1}$$

La probabilité $p_\varepsilon^{2:5}$ associée au second contexte 2 : 5 peut être obtenue avec (5.16) (page 107). Pour simplifier d'avantage la recherche pour le meilleur code,

l'assignation de les probabilités des symboles interdits dépendent de l'état, mais ne varient pas pour une valeur du contexte donnée. Cela réduit significativement le nombre d'automates à construire pour des valeurs des paramètres caractéristiques données qui deviennent maintenant $T = 32$, $f_{\max} = 1$, $\{p_0^1, p_1^1\}, \{p_0^{2:5}, p_1^{2:5}\}$, et un taux de codage souhaité $R_c = 14$ bits/symboles.

L'algorithme de Dijkstra est utilisé pour calculer la distance libre, et la méthode par tri est utilisée dans l'algorithme *branch-and-prune*. Le meilleur code à une distance libre de 6 et un taux de codage de $R_c = 13.9$ bits/symbole. Le temps nécessaire pour trouver la meilleure distance libre est de 1425 s.

La construction de CA conjoint pour de grands alphabets est donc possible en considérant une étape de binarisation. Cependant, la réduction du nombre de contextes et les contraintes imposées sur les symboles interdits conduisent à des CA conjoint qui sont nettement moins efficaces que ceux proposés par [But95], où un CLV conjoint pour $\mathcal{A}_{26}$ avec une distance libre de 5 et un $R_c = 10.41$ bits/symbole est proposé.

Des améliorations pourraient être obtenues en considérant plus de contextes et plus de variations sur l'assignation des probabilités des symboles interdits avec une dépendance vis-à-vis de l'état. Cependant le prix à payer est une grande complexité de calcul. L'algorithme de type *branch-and-prune* proposé peut être fortement parallélisé, ce qui pourrait aider à résoudre en partie ce problème.

6.1.3 Complexité

Il serait intéressant pour l'algorithme de type *branch-and-prune* de trouver une relation entre les paramètres caractéristiques et le temps nécessaire pour trouver le meilleur automate. Cependant, cela est très difficile puisque ce nombre dépend de ces paramètres de manière très complexe. On peut calculer une borne supérieure du nombre d'états par automate, ensuite une borne supérieure sur le nombre d'automates, mais cette borne sera probablement très faible. Une difficulté supplémentaire réside dans l'estimation du temps nécessaire par l'algorithme pour calculer la distance libre d'un automate. Ceci est aussi très difficiles à estimer à partir des paramètres caractéristiques sans construire réellement l'encodeur à états finis.

6.2 Codes à longueur variable type Huffman

Ce paragraphe présente trois ensembles d'expérience. D'abord, les différents algorithme de type *branch-and-prune* présentés au paragraphe 5.1 sont comparés à une recherche exhaustive pour des codes courts. Ensuite, l'évolution de la complexité d'un algorithme avec la taille du vecteur de Kraft est considérée. Finalement, la construction de CLVs conjoints pour des alphabets de grandes tailles est considérée.

Les trois méthodes d'exploration de l'arbre (en profondeur, en largeur et par tri) sont déjà comparées dans le paragraphe 6.1 et cette comparaison met en évidence que la méthode du tri est la meilleur pour trouver rapidement le code conjoint avec la plus grande distance libre. Pour toutes les simulations suivantes, le parcours de l'arbre par la méthode du tri sera utilisé.

Les expériences ont été effectuées sur un Intel Xeon E5420 à 2.50GHz et 64 GO de mémoire.

6.2.1 *Branch-and-prune* versus recherche exhaustive

Le premier vecteur de Kraft considéré est $\boldsymbol{\ell} = (4, 5, 6, 7)$. Le tableau 6.2 montre le temps nécessaire aux méthodes détaillées au paragraphe 5.1 pour trouver le CLV conjoint avec la plus grande distance libre. L'algorithme d'optimisation générique est utilisé avec une expansion des dictionnaires par plan de bits (*Bp*, voir paragraphe 5.3.1.2)) et par mot de code (*Cw*, voir paragraphe 5.3.1.1). Les représentations d'arbres canoniques sont ensuite considérées avec l'expansion des dictionnaires par plan de bits (*C-Bp*) et par mot de code (*C-Cw*), voir paragraphe 5.3.1.3. *#Dictionnaires* est le nombre de CLVs conjoints (in)complets atteints. *#Arbres* et *#Arbres explorés* sont les nombres de représentations des arbres respectivement considérés et explorés, c'est-à-dire qui sont réellement spécifiés. Le meilleur CLV conjoint obtenu par chaque méthode est aussi indiqué. Le tableau 6.2 montre que tous les algorithmes *branch-and-prune* sont beaucoup plus efficaces que la recherche exhaustive en ce qui concerne le temps de calcul (la meilleure méthode est 200 fois plus rapide) et le nombre de CLV conjoint examiné (plus de 200 fois moins de dictionnaires pour la meilleur méthode).

Si on considère le nombre de CLVs conjoints intermédiaires qu'il a fallu examiné, l'expansion par *mot de code* est beaucoup plus efficace, seule ou réalisée sur les

TABLE 6.2 – Comparaison entre la recherche exhaustive et les algorithmes de type *branch-and-prune* pour le vecteur de Kraft $\boldsymbol{\ell} = (4, 5, 6, 7)$. L'extension de la borne de Heller conduit à $\bar{d}^{\text{he}}_{\text{libre}} = 6$ pour $n = 10$.

Méthode	**Exhaustive**	**Bp**	**Cw**	**C-Bp**	**C-Cw**
#Dictionnaires	1 586 880	83 400	5 862	28 566	12 144
d_{libre}	6	6	6	6	6
Temps (s)	234	37	0.37	5	1
#Arbres	-	-	-	970	970
#Arbres explorés	-	-	-	108	108
4	0111	0001	0000	0111	0000
5	11001	11110	01111	11000	11110
6	000000	101011	110101	101011	101011
7	1001011	0100100	1001101	1001100	1001101

représentations des arbres canoniques. Avec cette méthode d'expansion, des bornes plus précises de la distance libre sont obtenues plus tôt lors du parcourt de l'arbre de recherche, ce qui permet de l'élaguer plus efficacement.

D'autres expériences montrent que C-Cw devient plus efficace que Cw lorsque les dictionnaires deviennent assez grands, plus spécifiquement pour des dictionnaires avec beaucoup de mots de code de même longueur. Par exemple, considérons le vecteur de Kraft $\boldsymbol{\ell} = (4, 5, 5, 6, 6, 7, 7)$. C-Cw examine 682322 CLVs conjoints en 746 s, tandis que Cw considère 1874127 CLVs conjoints en 3152 s pour obtenir les dictionnaires avec la même distance libre. Par conséquent, les expériences suivantes ont été réalisées avec C-Cw.

6.2.2 Complexité de C-Cw

Pour évaluer la complexité de la méthode de recherche par C-Cw, on doit d'abord estimer le nombre de représentations d'arbres canoniques, pour un vecteur de Kraft donné, et le nombre de dictionnaires préfixes que l'on peut construire pour chaque représentation.

Soit le vecteur de Kraft $\boldsymbol{\ell}$ pour M composants. Le nombre de représentations d'arbres canoniques $N_{\text{T}}(\boldsymbol{\ell})$ qui peut être obtenu avec $\boldsymbol{\ell}$ peut être évalué récursive-

ment comme :

$$N_{\mathrm{CT}}(\boldsymbol{\ell}) = \sum_{(\boldsymbol{\ell}_1,\boldsymbol{\ell}_2)\in\mathcal{V}(\boldsymbol{\ell}-1)} N_{\mathrm{CT}}(\boldsymbol{\ell}_1)\, N_{\mathrm{CT}}(\boldsymbol{\ell}_2) \tag{6.2}$$

où

$$\mathcal{V}(\boldsymbol{\ell}) = \{\boldsymbol{\ell}_1, \boldsymbol{\ell}_2 | \boldsymbol{\ell}_1 \cup \boldsymbol{\ell}_2 = \boldsymbol{\ell}, \boldsymbol{\ell}_1 \prec_{\mathrm{lex}} \boldsymbol{\ell}_2, K(\boldsymbol{\ell}_1) \leqslant 1, K(\boldsymbol{\ell}_2) \leqslant 1\}$$

contient toutes les paires de sous vecteurs $\boldsymbol{\ell}_1$ et $\boldsymbol{\ell}_2$ obtenues en subdivisant $\boldsymbol{\ell}$, de manière à ce que $\boldsymbol{\ell}_1 \prec_{\mathrm{lex}} \boldsymbol{\ell}_2$ et que des codes préfixes peuvent être obtenus avec les longueurs des mots de code contenues dans $\boldsymbol{\ell}_1$ et $\boldsymbol{\ell}_2$. Dans (6.2), $N_{\mathrm{CT}}(\emptyset) = N_{\mathrm{CT}}(0) = 1$.

Pour un choix donné de $\boldsymbol{\ell}_1$ et $\boldsymbol{\ell}_2$, le nombre de fois que $\boldsymbol{\ell}_1 - 1$ et $\boldsymbol{\ell}_2 - 1$ peuvent être subdivisés peut encore être borné (borne supérieure) par (6.2). Si M est assez grand, le nombre de représentation d'arbres canoniques devient rapidement très grand. Pour l'exemple, avec $\boldsymbol{\ell}_1 = (4, 5, 6, 7)$, on obtient $N_{\mathrm{CT}}(\boldsymbol{\ell}_1) = 166$, tandis que pour $\boldsymbol{\ell}_2 = (4, 5, 5, 6, 7, 7)$, on obtient $N_{\mathrm{CT}}(\boldsymbol{\ell}_2) = 252608$.

Supposons maintenant que Q est le nombre de noeuds dans la représentation d'un arbre canonique. Le nombre de dictionnaires préfixes est donné par 2^{Q-M}. Clairement, examiner tous les dictionnaires préfixes pour un vecteur de Kraft donné qui contient beaucoup de mots de code, devient très vite ingérable.

Le tableau 6.3 montre l'évolution de la complexité de l'algorithme C-Cw lorsque la taille de l'alphabet augmente et où les nouveaux symboles ont des mots de code de longueurs au moins égales à la plus grande longueur précédente. Il compare aussi la distance libre du meilleur code obtenu en utilisant le SAT-based approach présenté dans [AKS10]. L'approche que nous proposons a un temps de calcul nettement plus élevé. Néanmoins, puisque nous considérons la vraie valeur de la distance libre comme critère de recherche, des meilleurs codes peuvent être obtenue comme dans l'exemple présent.

6.2.3 CLVs conjoints pour des grands alphabets

Pour construire les CLVs conjoints pour des sources avec de grands alphabets (typiquement plus de 10 symboles), on introduit une heuristique pour réduire le nombre de représentations canoniques à explorer et par conséquent, le nombre de dictionnaires à examiner. En contre partie, la recherche devient sous-optimale.

TABLE 6.3 – Évolution de la complexité pour trouver le meilleur code comme fonction du vecteur de Kraft en utilisant [AKS10] et C-Cw

			Utilisant [AKS10]		Approche proposée				
$\boldsymbol{\ell}$	$\bar{d}^{\text{pe}}_{\text{libre}}$	$\bar{d}^{\text{he}}_{\text{libre}}$	d_{libre}	Temps (s)	d_{libre}	Temps (s)	#Dictionnaires	#Arbres explorés	#Arbres
$(3,7,8,9)$	∞	8	6	0.03	7	5	15,228	23	50
$(3,7,8,9,11)$	∞	7	5	0.12	7	56	45,672	26	209
$(3,7,8,9,11,12)$	∞	7	5	0.4	7	478	140,204	83	1,595
$(3,7,8,9,11,12,13)$	∞	7	5	0.4	7	857	76,416	32	6,578
$(3,7,8,9,11,12,13,14)$	∞	7	4	97	7	12,553	566,660	209	37,953
$(3,7,8,9,11,12,13,14,15)$	∞	7	4	339	7	131,302	1,039,980	419	501,782

Le nombre de représentations canoniques peut être réduit en ne considérant que les arbres canoniques qui maximisent la borne supérieure de la distance libre donnée au paragraphe 4.3.4 (page 4.3.4), c'est-à-dire, les arbres dans lesquels les mots de code de même longueur ont un préfixe commun le plus court possible. Une heuristique raisonnable pour assurer cela, lors de la subdivision des vecteurs de Kraft pendant la construction des arbres canoniques, est de garantir que les deux nouveaux vecteurs ont le même nombre de mots de code de longueurs égales (ou que ce nombre ne diffère que d'un). Ce choix peut être justifié avec la borne de Plotkin (4.2) (page 70) qui montre que la borne supérieure de la distance libre décroit avec le nombre de mots de code de même longueur.

La première expérience dans le tableau 6.4 considère les 26 symboles les plus probables de l'alphabet anglais ; X_{26}, avec le vecteur de Kraft $\boldsymbol{\ell} = (2@5, 2@6, 4@7, 8@8, 6@9)$, où $m_i@\ell_i$ signifie comme dans [But95], m_i mots de code de longueur ℓ_i. C-Cw trouve une distance libre maximale de 4 et la longueur moyenne des mots de code est $\ell^J_{\text{moy}} = 7.3375$ bits/symbole. Pour X_{26}, l'entropie est $H(X_{(26)}) = 4.1752$ bits/symbole. Le schéma tandem équivalent utilisant un CLV de type Huffman (qui code un symbole à la fois) suivit d'un code convolutif de taux 1/2 avec une longueur de contrainte 2 conduit à une longueur moyenne de codage $\ell^T_{\text{moy}} = 8.4090$ bits/symbole avec une distance libre de 4. Le code de Huffman a une longueur moyenne de codage de 4.2045 bits/symboles. Ainsi pour la même capacité de correction d'erreurs, le code conjoints permet un gain du taux de codage de l'ordre de $\ell^T_{\text{moy}} - \ell^J_{\text{moy}} = 1.0715$ bits/symbole comparé au schéma tandem considéré.

La seconde expérience dans le tableau 6.4 a été effectuée pour les 16 symboles les plus probable de l'alphabet anglais, X_{16}. Le vecteur de Kraft utilisé est $\boldsymbol{\ell} =$

$(2@6, 2@7, 4@8, 4@9, 4@10)$ conduisant à $d_{\text{free}} = 5$ et à $\ell^J_{\text{moy}} = 7.750$ bits/symbole. L'entropie $H(X_{(16)}) = 3.821$ bits/symbole. Le schéma tandem équivalent utilisant un CLV de type Huffman (qui code un symbole à la fois) suivi d'un code convolutif de rapport 1/2 et une longueur de contrainte de 3 conduit à une distance libre de 5 et une longueur moyenne globale de $\ell^T_{\text{moy}} = 2 \times H(X_{16}) = 7.705$ bits/symbole. Pour la même distance libre la perte du taux de codage est de $\ell^J_{\text{moy}} - \ell^T_{\text{moy}} = 0.113$ bits/symbole.

Dans tous les cas la borne $\bar{d}^{\text{he}}_{\text{libre}}$ coïncide avec la distance libre réelle du code obtenu, montrant ainsi son efficacité pour les alphabets à grande taille.

Il semble que la complexité de la recherche est dominée par le nombre d'évaluations de d_{libre} (les bornes) en utilisant l'algorithme de Dijkstra, dont la complexité avec l'implémentation courante est de $O(|\mathcal{S}_b|^4)$. Le temps d'exécution pour $X_{(26)}$ est plus petit que pour $X_{(16)}$ du au fait que pour $X_{(26)}$, un seul arbre a été exploré pour obtenir le code dont la distance libre coïncide avec la l'extension de la borne supérieure de Heller, alors que, quatre arbres ont été explorés pour $X_{(16)}$, nécessitant beaucoup plus d'évaluation de distances.

Pour un CLV conjoint avec le vecteur de Kraft $\boldsymbol{\ell}^J = \left(\ell^J_1, \ldots, \ell^J_M\right)$, la complexité pour le décodage peut être évaluée comme le nombre d'états dans le codeur à états finis synchronisé bit au niveau du décodeur, $S^J_b = \sum_{i=1}^{M} \ell^J_i - M + 1$.

Pour le schéma tandem équivalent composé d'un CLV de type Huffman avec $\boldsymbol{\ell}^T = \left(\ell^T_1, \ldots, \ell^T_M\right)$ suivi d'un code convolutif de taux $1/n$ avec la longueur de contrainte L_c, la complexité du décodeur conjoint dans le graphe produit $\Gamma^{\text{VLC}}_b \times \Gamma^{\text{CC}}_b$ peut être considérée, avec $S^T_b = S^{\text{CLV}}_b \cdot S^{\text{CC}}$ états, où $S^{\text{CLV}}_b = \sum_{i=1}^{M} \ell^T_i - M + 1$ et $S^{\text{CC}} = 2^{L_c - 1}$;

Pour l'exemple de la source $X_{(26)}$ considérée ci-dessus, $S^J_b = 197$ et $S^T_b = 468$ alors que pour la source $X_{(16)}$, $S^J_b = 119$ et $S^T_b = 104$.

Alternativement, on peut considérer des décodeurs séparés et mettre toute la complexité dans le CC. Les performances du décodeur du code convolutif seront probablement améliorés, cependant, la propagation d'erreur de décodage après le CLV de type Huffman sera plus couteuse que pour le schéma conjoint.

TABLE 6.4 – Construction de CLVs conjoints pour l'alphabet anglaise en utilisant C-Cw et l'heuristique proposée ; les probabilités utilisées sont celles de [But95]

Symboles	Probabilités	ℓ	Dictionnaire	ℓ	Dictionnaire
$a_1 = E$	$p_{a_1} = 0.1270$	5	00000	6	010110
$a_2 = T$	$p_{a_2} = 0.0906$	5	11110	6	101001
$a_3 = A$	$p_{a_3} = 0.0817$	7	0110000	7	0110101
$a_4 = O$	$p_{a_4} = 0.0751$	7	0011101	7	1001010
$a_5 = I$	$p_{a_5} = 0.0697$	7	1001000	8	00001100
$a_6 = N$	$p_{a_6} = 0.0674$	7	1100110	8	01100111
$a_7 = S$	$p_{a_7} = 0.0633$	8	01010101	8	11110000
$a_8 = H$	$p_{a_8} = 0.0609$	8	01110010	8	10011011
$a_9 = R$	$p_{a_9} = 0.0599$	8	00011000	9	001111011
$a_{10} = D$	$p_{a_{10}} = 0.0425$	8	00101110	9	011101100
$a_{11} = L$	$p_{a_{11}} = 0.0403$	8	10100101	9	110000111
$a_{12} = C$	$p_{a_{12}} = 0.0278$	8	10000010	9	100010000
$a_{13} = U$	$p_{a_{13}} = 0.0276$	8	11101000	10	0010000000
$a_{14} = M$	$p_{a_{14}} = 0.0241$	8	11011011	10	0111110011
$a_{15} = W$	$p_{a_{15}} = 0.0236$	10	0100101100	10	1101111100
$a_{16} = F$	$p_{a_{16}} = 0.0223$	10	0101110011	10	1000001011
$a_{17} = G$	$p_{a_{17}} = 0.0202$	10	0110110101		
$a_{18} = Y$	$p_{a_{18}} = 0.0197$	10	0111111000		
$a_{19} = P$	$p_{a_{19}} = 0.0193$	10	0000101011		
$a_{20} = B$	$p_{a_{20}} = 0.0149$	10	0001001101		
$a_{21} = V$	$p_{a_{21}} = 0.0098$	10	0011010100		
$a_{22} = K$	$p_{a_{22}} = 0.0077$	10	0010010011		
$a_{23} = J$	$p_{a_{23}} = 0.0015$	10	1011110010		
$a_{24} = X$	$p_{a_{24}} = 0.0015$	10	1010101101		
$a_{25} = Q$	$p_{a_{25}} = 0.0010$	10	1001110101		
$a_{26} = Z$	$p_{a_{26}} = 0.0007$	10	1000111000		
		$\ell_{\text{moy}} = 7.3375$	$d_{\text{libre}} = 4$	$\ell_{\text{moy}} = 7.750$	$d_{\text{libre}} = 5$
Bornes sur d_{libre}			$\overline{d}^{\text{he}}_{\text{libre}} = 4$		$\overline{d}^{\text{he}}_{\text{libre}} = 5$
Temps d'exécution			310 h		554 h
Arbres générés			3,130		43,172
Arbres explorés			1		4
d_{libre} évaluations			384,336		3,121,150

6.3 Conclusions

Dans ce chapitre, nous avons mené un ensemble d'expériences pour évaluer les performances des méthodes proposées dans cette thèse.

* Un premier ensemble d'expériences montre que pour la recherche de codes conjoints, la méthode (décrite au chapitre 3) utilisant l'algorithme de Dijkstra sur le graphe des distances entre paires est très efficace comparée aux méthodes décrites dans la littérature (en particulier celle dans [BJWK08]).
* Un second ensemble d'expériences montre que pour trouver le code conjoint avec la plus grande distance libre, l'algorithme de type *branch-and-prune* (voir le paragraphe 5.1) est nettement plus efficace qu'une recherche exhaustive. Pour le parcours de l'arbre des codeurs à états finis, les résultats montrent aussi que la méthode du tri est meilleur par rapport à l'exploration en largeur et l'exploration en profondeur.
* Des expériences ont d'abord été menées pour construire des codes arithmétiques conjoints pour une source à entrés binaires ($M = 2$), puis pour des sources non binaires, en particulier pour les 26 symboles les plus probables de l'alphabet anglais (X_{26}). Comparé au schéma tandem équivalent (CA suivi d'un code convolutif), pour le même taux de codage, le schéma conjoint obtenu reste sous-optimale, du à un manque de mémoire par rapport au code convolutif.
* Un dernier ensemble d'expériences a permis de construire des codes à longueur variable conjoints pour des alphabets courts (moins de 10 symboles) et pour des alphabets longs, plus précisément pour les 16 et les 26 premiers symboles de l'alphabet anglais (X_{16} et X_{26} respectivement). Comparé aux schémas tandem équivalent, c'est-à-dire (code d'Huffman suivi d'un code convolutif) avec la même distance libre, le code conjoint pour X_{16} introduit une perte en taux de codage et le code conjoint pour X_{26} permet un gain en taux de codage. Les expériences montrent aussi que les bornes sur la distance libre proposées, sont très efficaces pour les codes à longueur variable conjoints avec des grands alphabets.

L'ensemble des expériences montrent que la constructions de codes conjoints avec des alphabets de grandes tailles est possible avec l'approche que nous proposons

dans cette thèse. Néanmoins, les codes obtenus sont généralement sous optimaux par rapport aux codes tandems équivalents. Cette sous-optimalité est due au fait que les schéma conjoints que nous proposons ici manquent de mémoire par rapport au code convolutif pour aboutir à des distances libres équivalentes. Mais la complexité au niveau du décodeur des codes conjoints reste en général inférieure à la complexité du décodage conjoint du schéma tandem.

Chapitre 7

Conclusions et perspectives

7.1 Conclusions

Cette thèse avait deux objectifs principaux. Le premier consiste à proposer des outils analytiques pour caractériser les performances de correction d'erreurs (la distance libre et le spectre de distances) de codes source-canal conjoints (CSCCs) tels que les codes arithmétiques (CAs) conjoints et les codes à longueur variable (CLVs) conjoints. Le second consiste à exploiter ces outils pour proposer de nouvelles méthodes de construction de CSCCs afin d'optimiser leurs performances en terme de correction d'erreurs. Trois principales contributions ont été apportées dans cette thèse.

7.1.1 Évaluation de la distance libre d'un code conjoint

Dans cette thèse, j'ai étendu des méthodes de fonction de transfert sur graphe établies pour les codes de canal à taux fixe, pour calculer la distance libre et le spectre de distances des codes à longueur variables décrits par des machines à états finis (voir le chapitre 3). La méthode résultante pour le calcul de la distance libre utilise l'algorithme de Dijkstra sur le graphe des distances entre paires de suites de mots de code.

Cette méthode est beaucoup plus efficace que la technique réalisant une énumération de suite de mots de code et présentée dans [BJWK08] pour les CAs conjoints. Elle n'a en particulier pas de souci pour traiter les codes catastrophiques.

7.1.2 Bornes sur la distance libre

J'ai également introduit au chapitre 4 plusieurs nouvelles bornes sur la distance libre des codes conjoints. Ces bornes sont très utiles pour la construction de codes conjoints. Les bornes incluent une généralisation de la borne de Plotkin aux CLVs conjoints et une extension de la borne de Heller aux CLVs/CAs conjoints. Les expériences effectuées montrent que pour les CLVs conjoints avec des grands alphabets, les extensions des bornes de Plotkin et de Heller sont très efficaces.

Pour les CLVs conjoints, plusieurs bornes sont aussi obtenues en utilisant le graphe des distances entre paires de suites de mots de code, en considérant que tous les bits ne sont pas spécifiés dans le code considéré. Pour ces dernières bornes, la connaissance de la structure de l'arbre de code associé au dictionnaire permet d'obtenir des bornes moins conservatives.

7.1.3 Optimisation de la distance libre

La troisième contribution dans cette thèse a été de proposer de nouvelles méthodes utilisant un algorithme de type *branch-and-prune* pour l'optimisation de la distance libre des codes conjoints. Le code qui en résulte maximise la distance libre, et non sa borne inférieure comme dans la plupart des autres techniques d'optimisation de codes.

L'approche proposée consiste à organiser les codes conjoints dans une structure d'arbre à codes et parcourir cet arbre avec un algorithme de type *branch-and-prune* pour trouver le code conjoint avec la plus grande valeur de la distance libre. Les résultats des expériences montrent que l'algorithme de type *branch-and-prune* est beaucoup plus efficace qu'une recherche exhaustive.

Trois méthodes d'explorations de l'arbre ont été étudiées : l'exploration en profondeur, l'exploration en largeur et l'exploration glouton. Il en résulte que l'algorithme glouton est le plus rapide pour la recherche de codes avec la plus grande distance libre.

dans le cas des CLVs conjoints, pour un vecteur de Kraft $\boldsymbol{\ell}$ fixé, cette thèse décrit trois méthodes pour structurer les CLVs conjoints dans une structure d'arbre : la construction par mots de code, la construction par plan de bits et la construction par arbres canoniques. Cette dernière avec une exploration des arbres de codes par

mot de code est la plus efficace pour la recherche de code.

Dans le cas des codes arithmétiques conjoints, pour des valeurs de paramètres caractéristiques (précision du codeur, $f_{\max}$, probabilités de la source et la redondance) du codeur fixées, j'ai présenté une méthode pour structurer les CAs conjoints dans un arbre à codes. Cette méthode utilise la notion de symboles interdits multiples et attribue à chaque symbole interdit une probabilité qui dépend de l'état du codeur.

Les expériences effectuées montrent que la construction de codes conjoints de grandes tailles est possibles avec la méthode proposée dans cette thèse, mais cette construction reste complexe. On constate aussi que les codes conjoints obtenus restent sous-optimaux concernant le taux de codage et la distance libre par rapport des schémas tandems équivalents mais restent moins complexes en considérant le nombre d'états et de transitions au niveau du décodeur. Comme nous l'avons déjà précisé, cette sous-optimalité est due au fait que le schéma conjoint manque de mémoire par rapport à un code convolutif dont la capacité de correction d'erreurs augmente avec la taille de la mémoire.

7.2 Extensions de cette thèse

Depuis le début de la rédaction de cette thèse, plusieurs travaux sont en cours.

7.2.1 Formulation de la synthèse de CLVs conjoints comme un problème MILP avec des contraintes fournies par le GDP

Une première extension à cette thèse a été d'appliquer sur le graphe des distances entre paires (GDP) des algorithmes d'optimisation de type *mixed-integer linear programming* (MILP) pour la construction des CLVs conjoints optimaux. Ces travaux ont été menés en collaboration avec Hassan Hijazi et Leo Liberti du Laboratoire d'Informatique de l'École Polytechnique. Ces travaux ont abouti à une publication [HDK+12] en 2012 et sont brièvement décrits ci-dessous.

Comme nous l'avons déjà précisé aux paragraphes 2.2.1 et 4.3.3, à partir d'un vecteur de Kraft $\boldsymbol{\ell} = (\ell_1, \ell_2, \ldots, \ell_M)$ donné, on peut construire un dictionnaire totalement indéterminé $\mathcal{C}$ dans lequel tous les bits sont des étiquettes symboliques

c_i^j :

$$\mathcal{C} = \left\{ c_1^1 c_1^2 \cdots c_1^{\ell_1}, c_2^1 c_2^2 \cdots c_2^{\ell_2}, \cdots, c_M^1 c_M^2 \cdots c_M^{\ell_M} \right\}. \tag{7.1}$$

Les arcs orientés du graphe des distances entre paires (GDP) obtenus à partir de $\mathcal{C}$ seront étiquetés avec la somme modulo 2 d'étiquettes symboliques (voir la figure 4.3 à la page 77).

Soit $\mathcal{P}_{\mathrm{e}}$ l'ensemble des chemins élémentaires (sans cycle) dans le GDP qui partent de l'état s_{div} et arrivent à l'état s_{conv}. Soit d une distance de Hamming donnée ; selon la définition 2.14 (page 55) on a :

$$d_{\mathrm{libre}}(\mathcal{C}) \geqslant d \text{ si et seulement si } \forall\, \boldsymbol{e} \in \mathcal{P}_{\mathrm{e}}, w_{\mathrm{H}}(\boldsymbol{e}) \geqslant d. \tag{7.2}$$

Ainsi, pour un vecteur de Kraft donné et une distance de Hamming d donnée, trouver le CLV conjoint $\mathcal{C}$ de distance libre supérieure ou égale à d, revient à choisir les c_i^j dans $\{0,1\}$ qui permettent de satisfaire (7.2) dans le GDP et qui laissent le code $\mathcal{C}$ préfixe. Cette dernière condition (code préfixe) peut s'exprimer de la manière suivante

$$d_{\mathrm{div}}(\mathcal{C}) \geqslant 1, \tag{7.3}$$

où d_{div} est la distance minimale divergente du code (voir le paragraphe 4.2.3 à la page 70).

Pour un vecteur de Kraft donné, trouver le code avec la plus grande distance libre revient à trouver la plus grande valeur de d dans $\left\{1, 2, \cdots, \bar{d}^{\mathrm{he}}\right\}$ pour laquelle il existe un choix possible des c_i^j ($1 \leqslant i \leqslant M$ et $1 \leqslant j \leqslant \ell_M$) dans $\{0,1\}$ qui satisfait à la fois (7.2) et (7.3).

Ce problème a été abordé ici avec des outils d'optimisation de type MILP où les c_i^j ($1 \leqslant i \leqslant M$ et $1 \leqslant j \leqslant \ell_M$) sont les variables de décision, d est la variable à maximiser ; et (7.2) et (7.3) ont été traduites en contraintes d'inégalités linéaires.

Le meilleur CLV conjoint que nous obtenons pour les 26 symboles de l'alphabet anglais avec cette méthode a une distance libre de 7 et une longueur moyenne de $\ell_{\mathrm{moy}}^J = 10.11$ bits/symbole (voir le tableau 7.1). Le vecteur de longueur a été choisi de manière empirique, et d'éventuelles améliorations sont encore possibles. A notre connaissance, c'est la meilleure performance pour un CLV conjoint avec une distance libre de 7. Dans [LP03], une heuristique optimisée renvoie une solution avec une longueur moyenne de 10.738 bits/symbole.

TABLE 7.1 – CLVs conjoints pour l'alphabet anglais.

Symbole	Probabilité	ℓ	Dictionnaire	ℓ	Dictionnaire
$a_1 = E$	$p_{a_1} = 0.1270$	7	1010100	5	00011
$a_2 = T$	$p_{a_2} = 0.0906$	7	0101011	6	010101
$a_3 = A$	$p_{a_3} = 0.0817$	8	00011101	7	1111000
$a_4 = O$	$p_{a_4} = 0.0751$	9	111100010	8	10000100
$a_5 = I$	$p_{a_5} = 0.0697$	10	0010010011	8	00110010
$a_6 = N$	$p_{a_6} = 0.0674$	10	1100001000	8	11011011
$a_7 = S$	$p_{a_7} = 0.0633$	11	00100011010	8	01101101
$a_8 = H$	$p_{a_8} = 0.0609$	11	11011111000	9	101000000
$a_9 = R$	$p_{a_9} = 0.0599$	11	10110100101	9	110110001
$a_{10} = D$	$p_{a_{10}} = 0.0425$	11	01001000111	9	111111110
$a_{11} = L$	$p_{a_{11}} = 0.0403$	12	010110000000	9	100001111
$a_{12} = C$	$p_{a_{12}} = 0.0278$	12	100001101000	10	0010011000
$a_{13} = U$	$p_{a_{13}} = 0.0276$	12	011001010011	10	1000100100
$a_{14} = M$	$p_{a_{14}} = 0.0241$	12	001110111111	10	1110101010
$a_{15} = W$	$p_{a_{15}} = 0.0236$	13	0000000100000	10	1011001101
$a_{16} = F$	$p_{a_{16}} = 0.0223$	13	1100010001111	11	11000011010
$a_{17} = G$	$p_{a_{17}} = 0.0202$	13	1110111011000	11	01100111100
$a_{18} = Y$	$p_{a_{18}} = 0.0197$	13	0111101000101	11	11101110001
$a_{19} = P$	$p_{a_{19}} = 0.0193$	15	001100101000001	11	10101001110
$a_{20} = B$	$p_{a_{20}} = 0.0149$	15	111001000100010	12	000000011011
$a_{21} = V$	$p_{a_{21}} = 0.0098$	15	010000001001110	12	100010100000
$a_{22} = K$	$p_{a_{22}} = 0.0077$	15	110010111010001	12	110000101111
$a_{23} = J$	$p_{a_{23}} = 0.0015$	15	101101110101100	12	011011100111
$a_{24} = X$	$p_{a_{24}} = 0.0014$	15	001011101011010	12	110111010010
$a_{25} = Q$	$p_{a_{25}} = 0.001$	15	111110101110110	12	111010011001
$a_{26} = Z$	$p_{a_{26}} = 0.0007$	15	011011110000111	12	111111001100
		$\ell_{\text{moy}} = 10.11$	$d_{\text{libre}} = 7$	$\ell_{\text{moy}} = 8.158$	$d_{\text{libre}} = 5$
Bornes sur d_{libre}			$\overline{d}^{\text{he}}_{\text{libre}} = 7$		$\overline{d}^{\text{he}}_{\text{libre}} = 5$

Le schéma tandem de même distance libre utilisant un code de Huffman suivi d'un code convolutif de taux 1/2, de longueur de contrainte 5 a une longueur moyenne $\ell^T_{\text{moy}} = 8.4090$ bits/symbole (voir le paragraphe 115 à la page 6.2.3). Par rapport au schéma tandem, le code conjoint obtenu introduit une perte en compression de $\ell^J_{\text{moy}} - \ell^T_{\text{moy}} = 1.701$ bits/symbole. Par contre le nombre d'états dans le décodeur du schéma conjoint est de $S^J_b = 334$ et celui du décodeur conjoint du schéma tandem est de $S^T_b = 1872$ donc beaucoup plus complexe (voir le paragraphe 6.2.3 pour le détail des calculs).

Les expériences effectuées ont aussi permis de trouver un code CLV conjoint avec une distance libre de 5 et une longueur moyenne de 8.158 bits/symbole, ce qui est plus petit par apport aux 8.4752 bits/symbole trouvés dans [LP03]. Comparé à un schéma tandem de même distance libre utilisant un code convolutif de taux 1/2 avec une longueur de contrainte 3, le code conjoint obtenu permet un gain de compression $\ell^T_{\text{moy}} - \ell^J_{\text{moy}} = 0.251$ bits/symbole. De plus, $S^J_b = 229$ et $S^T_b = 468$. Donc pour la même distance libre, le schéma conjoint est plus efficace en compression et est moins complexe à décoder que le schéma tandem.

7.2.2 Synthèse de CLVs conjoints avec une contrainte sur la distance libre en utilisant un algorithme de type A*

Une autre suite proposée à cette thèse est l'utilisation d'un algorithme de type A* pour l'optimisation des codes à longueur variables conjoints. L'approche consiste à trouver pour un alphabet de taille M (avec le vecteur des probabilités) et une distance de Hamming d donnés, le code à longueur variable conjoint avec une distance libre de d et qui minimise la longueur moyenne du code. L'algorithme A* est utilisé et à la différence de [HWH10], les codes cherchés sont seulement préfixes et on calcule la vraie valeur de la distance libre à la place d'une borne sur la distance libre.

L'avantage de cette méthode est que la solution renvoyée sera l'optimale concernant le couple longueur moyenne et distance libre souhaitée.

Ces travaux sont en cours et sont effectués par un stagiaire Wenjie Li.

7.2.3 Amélioration du calcul de la distance libre

Le calcul de la distance libre présenté au chapitre 3 peut être fortement amélioré concernant le temps d'exécution en utilisant la machine à états finis du décodeur synchronisé bits à la place de la machine à états finis du codeur pour générer le GDP. En effet le graphe du décodeur à l'avantage d'avoir moins d'états que le graphe du codeur. Une amélioration supplémentaire pourrait être apportée en utilisant les *tas de Fibonacci* pour l'implantation de l'algorithme de Dijkstra.

Des premiers résultats apportés par un stagiaire, Aloïs Gruson, donnent les temps de calcul (en millisecondes) de la distance libre pour le premier code du tableau 7.1 ($d_{\text{libre}} = 7$) et pour les différentes améliorations :

- Dijkstra sur le GDP du codeur : 72176 ms
- Dijkstra avec les tas de Fibonacci sur le GDP du codeur : 561 ms
- Dijkstra avec les tas de Fibonacci sur le GDP du décodeur : 265 ms

D'autres résultats obtenus confirment aussi cette tendance.

7.3 Perspectives

Plusieurs perspectives s'offrent à cette thèse.

7.3.1 Code arithmétiques

Pour la construction de CAs conjoints, il serait intéressant de trouver un paramétrage alternatif à (précision du codeur,follow max, probabilités des symboles, redondance) qui permettrait de générer facilement la structure de l'automate sans l'étiquetage. Cela rendrait possible de calculer des bornes similaires aux extensions des bornes de Plotkin et de Heller pour les CLVs conjoints.

On peut considérer une extension des CAs conjoints avec une mémoire de m bits qui pourrait augmenter la distance libre en séparant les chemins qui conduisent à des petites distances. La mémoire stocke un entier $0 \leq \lambda \leq 2^m - 1$, ainsi l'état du codeur à états finis peut être représenté par (l, h, f, λ). L'ensemble des CAs conjoints avec une mémoire m contient l'ensemble des schémas tandem avec une longueur de contrainte $m + 1$. Par conséquent, on peut s'attendre à trouver au moins un CA conjoint avec des performances (taux de codage, distance libre) équivalentes à celles

du schéma tandem, mais moins complexe (en considérant le nombre d'états et de transitions).

7.3.2 Codes à longueur variables

On pourrait aussi considérer une extension pour les CLVs conjoints en introduisant une mémoire, par exemple en distinguant plusieurs états racine de l'arbre CLV, sans ou avec utilisation de codes distincts selon l'état de départ.

7.3.3 L'algorithme du *type branch-and-prune*

L'algorithme de type *branch-and-prune* introduit pourrait être fortement parallélisé pour un gain de temps considérable lors de la construction des codes.

Bibliographie

[AHU74] A. W. Aho, J. E. Hopcroft, and J. D. Ullman. *The design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[AKS10] N. Abedini, S. P. Khatri, and S. A. Savari. A sat-based scheme to determine optimal fix-free codes. *Proc. Data Compression Conference*, pages 169–178, 2010.

[Ast86] J. Astola. Convolutional code for phase-modulated channels. *Cybernetics and systems*, 17(1) :89–101, 1986.

[BCI+97] C. Boyd, J. Cleary, I. Irvine, I. Rinsma-Melchert, and I. Witten. Integrating error detection into arithmetic coding. *IEEE Transactions on Communications*, 45(1) :1–3, 1997.

[BF94] V. Buttigieg and P.G. Farrell. On variable-length error-correcting codes. In *Proceding IEEE International Symposium on Information Theory*, volume 147, page 507, 1994.

[BF00] V. Buttigieg and P.G. Farrell. Variable-length error-correcting codes. *IEEE Proceedings on Communications*, 147(4) :211–215, 2000.

[BHS06] D. Bi, W. Hoffman, and K. Sayood. State machine interpretation of arithmetic codes for joint source and channel coding. *Proc. of DCC, Snowbird, Utah, USA.*, pages 143–152, 2006.

[Big84] E. Biglieri. High-level modulation and coding for nonlinear satellite channels. *IEEE Transactions on Communications*, 32(5) :616–626, 1984.

[BJWK08] S. Ben-Jamaa, C. Weidmann, and M. Kieffer. Analytical tools for optimizing the error correction performance of arithmetic codes. *IEEE Transactions on Communications*, 56(9) :1458–1468, 2008.

[Bus97] S. R. Buss. Alogtime algorithms for tree isomorphism, comparison, and canonization. In *Proceedings of the 5th Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 18–33, London, UK, 1997. Springer-Verlag.

[But95] V. Buttigieg. *Variable-Length Error Correcting Codes*. Phd dissertation, University of Manchester, Univ. Manchester, U.K., 1995.

[CJ89] M. Cedervall and R. Johannesson. A fast algorithm for computing distance spectrum of convolutional codes. *IEEE transaction on Information Theory*, 35(6) :1146–1159, 1989.

[CR00] J. Chou and K. Ramchandran. Arithmetic coding-based continuous error detection for efficient ARQ-based image transmission. *IEEE Journal on Selected Areas in Communications*, 18(6) :861–867, 2000.

[CT91] T. M. Cover and J. M. Thomas. *Elements of Information Theory*. Wiley, New-York, 1991.

[DWK09] A. Diallo, C. Weidmann, and M. Kieffer. Optimazing the search of finite-states joint source-channel codes based on arithmetic coding. *Eusipco*, 2009.

[DWK10] A. Diallo, C. Weidmann, and M. Kieffer. Optimizing the free distance of error-correcting variable-length codes. *MMSP*, 2010.

[DWK11] A. Diallo, C. Weidmann, and M. Kieffer. Efficient computation and optimization of the free distance of variable-length finite-state joint source-channel codes. *IEEE Transactions on Communications*, 59(4) :1043–1052, 2011.

[DWK12] A. Diallo, C. Weidmann, and M. Kieffer. New free distance bounds and design techniques for joint source-channel variable-length codes. *IEEE Transactions on Communications*, 2012. Accepted for publication.

[Eli55] P. Elias. Coding for noisy channels. *IRE National Convention Record*, 3(4) :37–47, 1955.

[Eve64] S. Even. Test for synchronizability of finite automata and variable length codes. *IEEE Transaction on Information Theory*, 10(3) :185–189, 1964.

[FK90] A. S. Fraenkel and S. T. Klein. Bidirectional Huffman coding. *The Computer Journal*, 33(4) :296–307, 1990.

[For91] G. D. Forney. Geometrically uniform codes. *IEEE Transaction on Information Theory*, 37(5) :1241–1260, 1991.

[Gal68] R. G. Gallager. *Information Theory and Reliable Communication*. Wiley, New York, 1968.

[GM84] M. Gondran and M. Minoux. *Graphs and algorithms*. Wiley, Chichester, UK, 1984.

[HDK+12] A. Hijazi, A. Diallo, M. Kieffer, Leo. Liberti, and C. Weidmann. A milp approach for designing robust variable-length codes based on exact free distance computation. *Proc. of DCC, Snowbird, Utah, USA.*, pages 257–256, 2012.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9) :1098–1101, 1952.

[HV92] P. G. Howard and J. S. Vitter. Practical implementations of arithmetic coding. *Image and Text Compression*, 13(7) :85–112, 1992.

[HWH10] Y.-M. Huang, T.-Y. Wu, and Y.S. Han. An A*-based algorithm for constructing reversible variable length codes with minimum average codeword length. *IEEE Transactions on Communications*, 58(11) :3175 –3185, 2010.

[JZ99] R. G. Johannesson and K. Sh. Zigangirov. *Fundamentals of Convolutional Coding*. 1999.

[LP03] C. Lamy and J. Paccaut. Optimised constructions for variable-length error correcting codes. In *Proc. Information Theory Workshop*, pages 183–186, 2003.

[LV02] K. Lakovic and J. Villasenor. On design of error-correcting reversible variable length codes. *IEEE Communication Letters*, 6(8) :337–339, 2002.

[LV03] K. Lakovic and J. Villasenor. An algorithm for construction of efficient fix-free codes. *IEEE Communication Letters*, 7(8) :391–393, 2003.

[LWC08] C.-W. Lin, J.-L. Wu, and Y.-J. Chuang. Two algorithms for constructing efficient Huffman-code based reversible variable length codes. *IEEE Transactions on Communications*, 56(1) :81–89, 2008.

[Mas56] S.J. Mason. Feedback theory : Further properties of signal flow graphs. *Proceedings of the Institute of Radio Engineers*, 44(7) :920–926, July 1956.

[MH09] R. Maunder and L. Hanzo. Genetic algorithm aided design of component codes for irregular variable length coding. *IEEE Transactions on Communications*, 57(5) :1290 –1297, May 2009.

[MJG08] S. Malinowski, H. Jegou, and C. Guillemot. Error recovery properties and soft decoding of quasi-arithmetic codes. *EURASIP Journal on Advances in Signal Processing*, 2008(1) :1–12, 2008.

[MR85] J. C. Maxted and J. P. Robinson. Error recovery for variable length code. *IEEE Trans. Inform Theory*, 31(6) :794–801, 1985.

[Pas76] R. C. Pasco. *Source Coding Algorithms for Fast Data Compression.* PhD thesis, Stanford University, 1976.

[Pir82] Philippe Piret. Comma free error correcting code of variable length, generated by finite-state encoder. *IEEE Transaction on Information Theory*, 28(5) :764–775, 1982.

[Plo60] M. Plotkin. Binary codes with specified minimum distance. *IRE Transactions Information Theory*, 6(4) :445–450, 1960.

[Pro95] J.G. Proakis. *Digital Communications.* McGraw-Hill, Singapore, 1995.

[Ric03] I. E. G. Richardson. *H.264 and MPEG-4 Video Compression : Video Coding for Next-Generation Multimedia.* John Wiley and Sons, 2003.

[Ris76] J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3) :198–203, 1976.

[Sav09] S.A. Savari. On optimal reversible-variable-length codes. In *Information Theory and Applications Workshop*, pages 311 –317, 2009.

[Say99] J. Sayir. Arithmetic coding for noisy channels. *Proc. IEEE Information Theory Workshop*, pages 69–71, 1999.

[Say05] K. Sayood. *Introduction to data compression.* Elsevier, 2005.

[Sha48] C. E. Shannon. A mathematical thoery of communication. *Bell Syst. Tech. J.*, 27 :379–423 and 623–656, 1948.

[TC03] H.-W. Tseng and C.-C. Chang. Construction of symmetrical reversible variable length codes using backtracking. *The Computer Journal*, 46(1) :100–105, 2003.

[TK09] R. Thobaben and J. Kliewer. An efficient variable-length code construction for iterative source-channel decoding. *IEEE Transactions on Communications*, 57(7) :2005 –2013, 2009.

[TW01] Chien Wu. Tsai and Ling. Wu, Ja. On-constructing the huffman-codes based reversible variable-length codes. *IEEE Transactions on Communications*, 49(9) :1506–1509, 2001.

[TWM95] Y. Takishima, M. Wada, and M. Murakimi. Reversible variable length codes. *IEEE Transactions on Commununications*, 43(2/3/4) :158–162, 1995.

[Ung82] G. Ungerboeck. Channel coding with multilevel/phase signals. *IEEE Transactin on Information Theory*, 28(1) :55–67, 1982.

[Vit71] A. J. Viterbi. Convolutional codes and their performance in communication systems. *IEEE Transactions on Communications Technology*, 19(5) :751–772, 1971.

[VO79] A. J. Viterbi and J.K. Omura. *Principles of Digital Communication and Coding*. McGraw-Hill, New-York, 1979.

[WK10] C. Weidmann and M. Kieffer. Evaluation of the distance spectrum of variable-length finite-state codes. *IEEE. Transaction on Communication*, 58(3) :724–728, March 2010.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6) :520–540, 1987.

[WYH04] J. Wang, L.-L. Yang, and L Hanzo. Iterative construction of reversible variable-length codes and variable-length error-correcting codes. *IEEE Communications Letters*, 8(11) :671– 673, 2004.

[ZAC06] Y. Zhong, F. Alajaji, and L. L. Campbell. On the joint source-channel coding error exponent for discrete memoryless systems. *IEEE Trans. Information Theory*, 52(4) :1450–1468, 2006.

[ZW87] E. Zehavi and J. K. Wolf. On the performance evaluation of trellis codes. *IEEE Transaction on Information Theory*, 33(2) :196–202, 1987.